

**Forschungsberichte
der Fakultät IV – Elektrotechnik und Informatik**

**17th Edition of
ECOOP Doctoral Symposium
and PhD Workshop**

Proceedings

July 30, 2007, TU Berlin, Germany

Danny Dig (Ed.)

Bericht-Nr. 2007 – 7

ISSN 1436-9915

Table of Contents

ECOOP Doctoral Symposium and PhD Workshop Organization	iii
---	------------

Doctoral Symposium	
---------------------------------	--

- **Refactoring-Based Support for Binary Compatibility
in Evolving Frameworks**3
Ilie Savga and Uwe Aßmann (Technische Universität Dresden)
- **An Integrated Quantitative Assessment Model for
Usability Engineering**7
Haidar S Jabbar, Gopal T. V., and Sattar J Aboud (Anna University)
- **The Design and Implementation of Formal Monitoring Techniques.....**11
Eric Bodden (McGill University)
- **Flexible Ownership Domain Annotations for
Expressing and Visualizing Design Intent**15
Marwan Abi-Antoun and Jonathan Aldrich (Carnegie Mellon University)
- **Modelling Change-based Software Evolution.....**19
Romain Robbes and Michele Lanza (University of Lugano)
- **A Rewriting Approach to the Design and Evolution
of Object-Oriented Languages**23
Mark Hills and Grigore Rosu (University of Illinois at Urbana-Champaign)

PhD Workshop	
---------------------------	--

- **Checking Semantic Usage of Frameworks**29
Ciera Christopher Jaspan (Carnegie Mellon University)
- **An Integrated Method based on Multi-Models and Levels of Modeling for
Design and Analysis of Complex Engineering Systems**39
Michel dos Santos Soares and Jos Vrancken (Delft University of Technology)
- **Ordering Functionally Equivalent Software Components**49
Giovanni Falcone and Colin Atkinson (University of Mannheim)

ECOOP Doctoral Symposium and PhD Workshop Organization

Chair and Organizer: Danny Dig, University of Illinois at Urbana-Champaign

Program Committee: Danny Dig, University of Illinois at Urbana-Champaign
Jacqueline McQuillan, National University of Ireland
Naouel Moha, University of Montreal
Javier Perez, Universidad de Valladolid
Mikhail Roshchin, Volgograd State Tech University

Additional Reviewers: Paul Adamczyk, University of Illinois at Urbana-Champaign
José Manuel Marqués Corral, University of Valladolid
Foutse Khomh, University of Montreal
Guillaume Langelier, University of Montreal
Miguel Ángel Laguna Serrano, University of Valladolid
Manuel Barrio Solórzano, University of Valladolid
Mircea Trofin, Microsoft
Stéphane Vaucher, University of Montreal

We would like to thank AITO for its generous funding and its mission to support the Object technology.



Doctoral Symposium

Refactoring-Based Support for Binary Compatibility in Evolving Frameworks

Ilie Şavga and Uwe Aßmann (supervisor)

Institut für Software- und Multimediatechologie, Technische Universität Dresden,
Germany, {ilie.savga|uwe.assmann}@tu-dresden.de**

Abstract. The evolution of a software framework may invalidate existing plugins—modules that used one of its previous versions. To preserve *binary compatibility* (i.e., plugins will link and run with a new framework release without recompilation), we automatically create an adaptation layer that translates between plugins and the framework. The creation of these adapters is guided by information about syntactic framework changes (*refactorings*). For each refactoring that we support, we formally define a *comeback*—a refactoring used to construct adapters. For an ordered set of refactorings that occurred between two framework versions, the backward execution of the corresponding comebacks yields the adaptation layer.

1 Problem Description

Frameworks are software artifacts, which evolve considerably. A framework may change due to new requirements, bug fixing, or quality improvement. As a consequence, existing plugins may become invalid; that is, their sources cannot be recompiled or their binaries cannot be linked and run with a new framework release. Either plugin developers are forced to manually adapt their plugins or framework maintainers need to write update patches. Both tasks are usually expensive and error-prone. When the application has been delivered to a customer, it even may be undesirable to require plugin recompilation.

To analyze the nature of the client-breaking changes, Dig and Johnson [8] investigated the evolution of four big frameworks. They discovered that most (from 81% up to 97%) of such changes were *refactorings*—behavior-preserving source transformations [14]. The reason why pure structural transformations break clients is the difference between how a framework is refactored and how it is used. For refactoring it is generally assumed, that the whole source code is accessible and modifiable (the *closed world* assumption [8]). However, the frameworks are used by plugins not available at the time of refactoring. As a consequence, plugins are not updated correspondingly.

The existing approaches overcoming the closed world assumption of component evolution can be divided into two groups. Approaches of the first group

** The presented work is funded by the Sächsische Aufbaubank, project number 11072/1725.

rely on the use of a kind of legacy middleware (e.g., [3], [2], [13], [5]) or, at least, a specific communication protocol ([12], [9]) that connect a framework with its plugins. This, in turn, implies a middleware-dependent framework development, that is, the framework and its plugins must use an interface definition language and data types of the middleware and obey its communication protocols.

The second group consists of approaches to distribute the change specifications and to make them available for the clients remotely. The component developer has to manually describe component changes either as different annotations within the component’s source code ([4], [6], [15]) or in a separate specification ([11]). Moreover, the developer must also provide adaptation rules, which are then used by a transformation engine to adapt the old application code. Writing annotations is cumbersome and error-prone. To alleviate this task, the “catch-and-replay” approach [10] records the refactorings applied in an IDE as a log file and delivers it to the application developer, who “replays” refactorings on the application code. Still, current tools do not handle cases when a refactoring cannot be played back in the application context. For example, they will report a failure, if the renaming of a component’s method introduces a name conflict with some application-defined method.

2 Goal Statement

Our goal is to decrease the costs of plugin update in case of framework evolution. As a framework evolves, it should not break existing applications that conform to the Application Programming Interface of its previous releases. More specifically, we want to achieve automatic *binary compatibility* of framework plugins—existing plugins must link and run with new framework releases without recompiling [9].

Our solution is to automatically create, upon the release of a new framework version, an adaptation (wrapper) layer between the framework and plugins. The generation of adapters is guided by information about the refactorings that occurred between framework releases. The adapters then shield the plugins by representing the public types of the framework old version, while delegating to the new version.

We focus on the following research issues and questions:

- Demarcate the adaptation context. What are the limitations of the proposed technology? Can any possible refactoring (e.g., pure deletion of functionality) be supported? If no, what is a basic set of supported refactorings?
- Relate to existing work. How can we build our work using existing formal approaches for refactoring specifications? How easy can we integrate our approach with existing refactoring tools?
- Estimate performance and scalability. What is the performance overhead introduced by adaptation? Which programming languages and development platforms and to which extent can be supported?

We will answer these questions and provide tool support as a result of a project performed in collaboration with our industrial partner Comarch [1].

3 Approach

Basic Assumptions. Our main assumption is that the information about refactorings occurred between framework releases is available. This information can be obtained, for instance, by recording the applied refactorings in IDE ([10]) or detecting them in the source code (e.g. [17], [7]). Moreover, we assume that the order of refactoring execution is known.

Automated Adaptation. For each supported refactoring we formally define a *comeback*—a behavior-preserving transformation that defines how a compensating adapter is constructed. Because comebacks are formally defined, they serve as input specifications for an adapter generator. For an ordered set of refactorings that occurred between two framework versions, the execution of the corresponding comebacks in the reverse order yields the adaptation layer.

A comeback is defined in terms of refactoring operators to be executed on adapters. For some refactorings, the corresponding comebacks are simple and defined by a single refactoring. For example, to the refactoring *RenameClass* (*name*, *newName*) corresponds a comeback consisting of a refactoring *RenameClass* (*newName*, *name*) that renames the adapter to the old name. For other refactorings, their comebacks consist of sequences of refactorings. For instance, the comeback of *MoveMethod* is defined by *DeleteMethod* and *AddMethod* refactoring operators, which sequential execution effectively move the method between the adapters. A detailed description of our approach including formal comeback definition and current results is presented in [16].

Tool Support. Due to the requirements of our industrial partner, the tool is being implemented as a stand-alone application supporting .NET-based applications. Nevertheless, we plan to integrate it into an IDE and to evaluate our adaptation approach for Java applications, too. The most important tool requirements are the number of supported refactorings and the tool’s extensibility with new comebacks.

Evaluation. We will apply the tool to maintain both a big framework of our partner and a medium-size framework used for teaching at our university. We will evaluate how the use of our tool fosters framework maintenance by changing the type and number of applied API refactorings comparing to existing restrictive approaches. We will also perform benchmarking to estimate the performance penalties introduced by adaptation and the percentage of the supported modification operators (*recall*). Finally, we will test adapters and compare the results with the bug reports in case of manually implemented patches.

Discussion. Besides information about the public types and the refactoring history, our approach needs no additional component specifications. However, the latter are required to support modifications that go beyond refactoring (e.g., protocol changes). To adapt a broader range of changes, we will investigate and propose how to combine other adaptation techniques with our refactoring-based approach.

References

1. Comarch homepage. <http://www.comarch.com>.
2. CORBA homepage. <http://www.corba.org>.
3. Microsoft COM homepage. <http://www.microsoft.com/Com/default.mspx>.
4. I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 265–279, New York, NY, USA, 2005. ACM Press.
5. J. Camara, C. Canal, J. Cubo, and J. Murillo. An aspect-oriented adaptation framework for dynamic component evolution. In *3rd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 59–71, 2006.
6. K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 359, Washington, DC, USA, 1996. IEEE Computer Society.
7. D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In D. Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 404–428. Springer, 2006.
8. D. Dig and R. Johnson. The role of refactorings in API evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.
9. I. R. Forman, M. H. Conner, S. H. Danforth, and L. K. Raper. Release-to-release binary compatibility in SOM. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 426–438, New York, NY, USA, 1995. ACM Press.
10. J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support API evolution. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 274–283, New York, NY, USA, 2005. ACM Press.
11. R. Keller and U. Hölzle. Binary component adaptation. *Lecture Notes in Computer Science*, 1445:307–329, 1998.
12. S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic java classes. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 337–361, London, UK, 2000. Springer-Verlag.
13. F. McGurran and D. Conroy. X-adapt: An architecture for dynamic systems. In *Workshop on Component-Oriented Programming, ECOOP, Malaga, Spain*, pages 56–70, 2002.
14. W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.
15. S. Roock and A. Havenstein. Refactoring tags for automatic refactoring of framework dependent applications. pages 182–185, 2002.
16. I. Savga and M. Rudolf. Refactoring-based adaptation for binary compatibility in evolving frameworks. In *Proceedings of the Sixth International Conference on Generative Programming and Component Engineering*, Salzburg, Austria, October 2007. To appear.
17. P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

An Integrated Quantitative Assessment Model for Usability Engineering

Haidar S Jabbar, Gopal T V (advisor) and Sattar J Aboud

¹Department of Computer Science and Eng., Anna University, Chennai 600025, INDIA
haidar_sattar@yahoo.com, gopal@annauniv.edu

²Middle East University for Graduate Studies, Faculty of IT, Amman, Jordan
sattar_aboud@yahoo.com

Abstract. Many different quantitative usability assessment models have been proposed to measure and report the usability of a general software product. However, there are a number of problems associated in existing models that often result in subjective or biased usability assessments. Therefore, an integrated, consolidated and quantitative usability assessment model is required, to provide an entire construct of usability and to provide an improved estimation of the usability sample size. In this research, we propose an Integrated Quantitative Assessment Model for Usability Engineering (IQAMUE) for measuring and reporting usability, which improves the usability assessment measurement process at least in 3 points: (1) a broader integration of general potential usability factors and metrics, (2) employing statistical methods for normalizing the calculated metrics and (3) an improved estimation of the usability assessment sample size. Together the theoretical work, the conducted empirical case studies and the simulator tool developed, indicate that the proposed model effectively resolves and integrates major research problems found in the literature.

Keywords: Usability Assessment, Motivation of the Model, IQAMUE, Case Study, Sample Size Simulator

1 Problem Description

With the increase demand for software products and websites, the focus on the usability for these needs becomes essential [1]. Moreover, usability is increasingly recognized as a vital quality factor for interactive software systems, since most researchers agree that unusable systems are probably the single largest reasons why encompassing interactive systems, computers plus people, fail in their actual use [2].

Generally, the quality of any interactive system can be measured in terms of its usability [3]. However, it is often noticed that people spend huge amounts of money on eye-catching design for a software product rather than spending comparatively less to check this quality. This clarifies the growing numbers of research work in the literature that have devoted to the problem of how to measure and report usability. Several models have been proposed for measuring and reporting usability, however,

existing models have problems, which bias the measurement process and their results of usability. Below is a summary list of major research problems, which also are the motivation of our research work:

- Imprecise (small or high) number of users selected for usability assessment [6].
- Current models are not homogenous.
- Current models only include single factors of usability [3].
- Current models are measured on different scales [5].
- They are difficult to use and to communicate.
- Results are reported individually.
- The interpretation of usability is often not precise and clear.

2 Goal Statement

In this research work, the focus of the contribution to usability, has been given into three main issues, as described below.

Usability Definitions and Factors: As mentioned earlier, usability has not been defined in a homogenous way across the standards bodies or usability researchers. Most of these various models do not include all major factors of usability. They are also not well integrated into a single model. Therefore, the first step in IQAMUE was the investigation into existing models that represents usability factors, either by standard bodies such as ISO and ANSI or by well-known researches in the field of usability. Those various definitions from different resources, shows the confusion in the literature describing usability, which emphasize the need for a comprehensive usability assessment model. In the same time, the majority of models emphasizes the needs for the general factors of Efficiency, Effectiveness, Satisfaction, Memorability, Learnability, however they are not a well incorporated into existing models.

A Standardized Process: As discussed early, the methods and models of measuring and assessing usability can be divided into two broad categories: those that gather data from actual users and those that can be applied without actual users [4]. However, all these approaches assess usability independently. Individual measures do not provide a complete view of the usability of software product.

These metrics are typically reported individually on a task-by-task basis, with little context for interpreting the correlation among the metrics of a particular task or across a series of tasks. Therefore, we need to standardize the measurement process by using statistical methods to scale deviated measure and bring them into the same scale. We have used statistical methods to standardize the metrics, which have been already described and used in the literature many by researchers in [5].

Usability Assessment Sample Size: Another crucial problem in existing models and also a very important aspect for any usability assessment model is the estimation of the sample size desired for a software product. Once we start estimating the sample size needed for a usability assessment, a baffling question comes in mind: “how many users are enough for a given software product?”. Throughout the literature, sample size estimation for usability assessment has been done either as simply guessing, or using a mathematical formula proposed by the researchers. A variety of international

research work has been done on this topic, especially by the researchers: Virzi, Nielsen and Launder and Lewis [6].

To the best of our knowledge, the majority of existing models estimates the sample size needed for a usability assessment, based on historical data (previous software products). A better estimation should be based on both historical data, which provide an initial idea from previous software products and based on present data, which provide a practical idea for a given software product, which aid us in predicting the complexity of the software product by conducting a simple pre-assessment.

2 Approach

An Integrated Quantitative Assessment Model for Usability Engineering: (Usability Definitions and Factors): As discussed above that there are many definitions, which confuse the definition of usability and the use of these factors in the models. However, for our proposed model we have selected the factors: Efficiency, Effectiveness, Satisfaction, Memorability and Learnability.

We have selected these factors for the reasons (1) there are often cited by most publication in the literature (2) there is a common agreement of these factors in the literature based on standard bodies and on usability researchers (3) since we are measuring general software products, we find that those proposed factors used in our IQAMUE, falls in the general nature of the software products. However, other usability factors may be plugged in future versions of the proposed model.

A Standardized Process: Now the second part of the IQAMUE is to standardize the calculated metrics into scalable values. We have employed these methods to complement the measurement process by standardizing the values into a standardized scale. To standardize each of the usability metrics we need to create a normal score type value or z-equivalent. For the continuous and ordinal data (time, Learnability and satisfaction), we will subtract the mean value from a specification limit and divided by the standard deviation. For discrete data (task accomplishment, Memorability and errors) we will divide the unacceptable conditions (defects) by all opportunities for defects.

Now after calculating the z-scores for all the proposed metrics, we can effectively compare the measurements of the values, since all the values are now set on a standard percentage of (100%) scale. As discussed above, this type of model is not intended for collaborative assessment, it is intended to assess the usability of the software product at the final stages (pre release) with representative users; it is an independent assessment model, which could be applied only after completing the software product.

Usability Assessment Sample Size: Within the sample size of the usability assessment, we have proposed an improved and normalized estimation model for better estimation of the sample size. The proposed model enhances the estimation process, by using historical data to gain an initial idea of the software product, and on present data to predict the complexity of the software product, which is described below:

- Estimating the historical problem discovery rate (λ_a), which is already recommended by the researchers.

- Estimating the adjustment (vertical / domain wise) problem discovery rate (λ_p), by conducting a pre-assessment usability study, to gain a practical idea about the complexity for a given software product.

Integrating both points, gave us an improved and normalized estimation model toward the sample size for the usability assessment studies, where Alpha factor (α) is the initial historical estimation value based on λ_a and Beta factor (β) is the adjustment (vertical / domain wise) value based on λ_p . α is estimated from the historical problem discovery rate either taken as a constant (0.31) or from a set of individual unique problem discovery rates of similar software products (indicated in point 1 above) and β is estimated from the specific discovery rate for a given software product (indicated in point 2 above).

α is already explained in numerous research works mainly by Nielsen, Virizi and Turner and has been discussed early. However, this value is estimated from historical data λ_a (problem discovery rates). λ_a is either taken (0.31) as suggested by Nielsen [7] or could be estimated from previous historical data to replace the above value of λ_a . If we go for the historical data, then λ_a is estimated by creating a matrix of users' numbers and their related individual problem discovery rates.

β value is an adjustment factor, used to provide us crucial information related the sample size for a specific software product. λ_p is estimated by conducting a pre-assessment study of the highest task time of the software product. We have selected the (task time) to represent the software product complexity, because time always measures the whole complexity among other usability metrics. For computing β factor, we first start to conduct a pre-assessment usability study for the highest task time for (2-3) users. We need to keep in mind that those 2-3 users should differ in their experience, at least 1 novice and 1 experienced user, to estimate the maximum variance of time among users. For the applicability purpose of the model, we have conducted a case study to exercise the model. In addition we have developed a sample size estimation program to simulate the calculations of usability the sample size.

References

1. Rubin J., Handbook of Usability Testing: How to Plan, Design and Conduct Effective Tests. John Wiley and Sons (1994).
2. Haidar S. Jabbar and T. V. Gopal, User Centered Design for Adaptive E-Learning Systems. Asian Journal of Information Technology, 5 (2006) 429-436.
3. A. Seffah, Mohammad D., Rex B. Kline and Harkirat K. Padda, Usability Measurement and Metrics: A Consolidated Model. Software Quality Journal. 14 (2006) 159-178.
4. Patrick W. Jordan. An introduction to usability. Taylor and Francis (1998).
5. Sauro J. and Kindlund E. A Method to standardize Usability Metrics into a Single Score. In Proc. of the Conference in Human Factors in Computing Systems, Portland, (2006) 401-409.
6. Turner C. W., Lewis J. R. and Nielsen J., Determining Usability Test Sample Size. International Encyclopedia of Ergonomics and Human Factors, 3 (2006) 3084-3088.
7. Nielsen J., Why You Only Need to Test with 5 Users, retrieved from: <http://www.useit.com/alertbox/20000319.html>.

The design and implementation of formal monitoring techniques

Eric Bodden

Sable Research Group, School of Computer Science, McGill University
 eric.bodden@mail.mcgill.ca

Abstract. In runtime monitoring, a programmer specifies a piece of code to execute when a trace of events occurs during program execution. Previous and related work has shown that runtime monitoring techniques can be useful in order to validate or guarantee the safety and security of running programs. Yet, those techniques have not yet been able to make the transition to everyday use in regular software development processes. This is due to two reasons. Firstly, many of the existing runtime monitoring tools cause a significant runtime overhead, lengthening test runs unduly. This is particularly true for tools that allow reasoning about single objects, opposed to classes. Secondly, the kind of specifications that can be verified by such tools often follow a quite cumbersome notation. This leads to the fact that only verification experts, not programmers, can at all understand what a given specification means and in particular, whether it is correct. We propose a methodology to overcome both problems by providing a design and efficient implementation of expressive formal monitoring techniques with programmer-friendly notations.

1 Problem Description

Static program verification in the form of model checking and theorem proving has in the past been very successful, however mostly when being applied to small embedded systems. The intrinsic exponential complexity of the involved algorithms makes it hard to apply them to large-scale applications. Runtime monitoring or runtime verification [1] tries to find new ways to support automated verification for such applications. This is done by combining the power of declarative safety specifications with automated tools that allow to verify these properties not statically but dynamically when the program under test is executed. Researchers have produced a variety of such tools over the last years, many of which have helped to find real errors in large-scale applications.

Yet, those tools have not yet had any widespread adoption by programmers in real software development processes. In our opinion, this is mainly due to two reasons. Firstly, there is an obvious trade-off between expressiveness and complexity of any given runtime monitoring tool. The early tools were very lightweight, allowing users to specify properties such that *the method File.read(..) must be called only before File.close().* Such properties can be checked very efficiently. However, in an object-oriented setting, usually only per-object specifications

make sense: *for all Files f , $f.read(..)$ must be called only before $f.close()$* . Implementing runtime monitoring for such properties efficiently is a real challenge and until now the few tools which allow for such kind of specifications still induce an unduly large runtime overhead in many cases [4].

Secondly, many runtime verification tools build up on the mental legacy of static verification. In static verification, formalisms such as Linear Temporal Logic (LTL) and Computational Tree Logic (CTL) are very common. Even experts in formal verification admit that those formalisms are often hard to handle. Hence, it is only natural that many programmers perceive runtime monitoring as complicated and consequently not very practical. In addition, even if one programmer decides to go through the process of learning such a specification language, he might not be able to communicate specifications he wrote to any of his colleagues, making potential mistakes harder to spot.

This lack of adoption of runtime monitoring techniques therefore leads to the fact that in most software development projects formal verification simply does not take place. Instead, hand-written tests are produced; a process which in itself is tedious and error-prone. As a consequence, the potential of those powerful techniques just remains unused, leaving many faults, which otherwise could have been detected, quietly buried in program code.

2 Goal Statement

Our goal is to evolve the techniques of runtime monitoring to such a state that they can *easily be used by reasonably skilled programmers on large-scale applications* written in *modern programming languages*. Specifically, we want to tackle the problems of (1) efficient runtime monitoring for parametrized specifications and (2) providing specification formalisms that can easily be understood by programmers and can be used to not only verify runtime behaviour but also communicate design decisions between developers.

To ease the software development process, our approach should be automated as much as possible. Therefore, such a tool chain would have to consist of the following components:

- A front-end that provides support for denoting safety properties in a variety of (potentially graphical) specification formalisms.
- A generic back-end that allows for the automatic generation of runtime monitors for any such formalism. The generated monitors should be as efficient as possible, even if specifications are to be evaluated on a per-object basis.
- A static analysis framework to specialize instrumentation code with respect to the program under test. Goal of this framework is to remove any instrumentation overhead induced by the monitor, in case this overhead can statically be proven unnecessary.

This tool chain would address the stated problems in the following ways. The potentially graphical front-end would allow programmers to denote safety properties in a way that is close to their mental picture of it. Bridging this gap

between what the programmer wants to express and needs to express is essential in order to guarantee a minimal chance for error at specification time. The front-end should potentially be integrated into an existing integrated development environment in order to provide programmers easy access.

The back-end then generates efficient code from those safety properties. At this stage, high-level events are mapped onto events in the actual code base. From our experience we can tell that one potentially good way of doing this would be to use *pointcuts* of an aspect-oriented programming language [6]. Such pointcuts have proven themselves to be easy enough to understand for many average software developers, as their wide-spread use in software development proves.

Although the back-end generates efficient code, this code might still not be efficient enough, in particular in scenarios where the program under test would trigger the generated runtime monitor very frequently. Static program analysis can decrease the runtime overhead by statically determining that certain static instrumentation points can never be part of a dynamic trace that would trigger a violation of the given specification. The static analysis back-end should be able to offer generic analyses that can be applied to specifications in arbitrary formalisms and flexible enough to allow additional formalism-specific analysis stages to be plugged-in. Furthermore it must be capable of automatically specializing the generated monitor based on the gathered analysis results. In order to make sure the overall goal is reached, we propose the following methodology.

3 Approach

Efficient monitor code generation: We base our approach on an already developed back-end for *tracematches* [2]. Tracematches are an extension to the aspect-oriented programming language AspectJ [3] which allows programmers to specify traces via regular expressions with free variables. Avgustinov et al. already identified and solved many of the problems of generating efficient monitor code [4], yet for some benchmarks large overheads remain.

Removal of unnecessary instrumentation through static analysis: In a second step (ongoing), we then design and implement a set of static analyses which allows us to remove unnecessary instrumentation induced by the presence of tracematches. Initial results seem promising, lowering the runtime overhead to under 10% in most cases [5].

Making code generation and analysis generic: In a third step we then plan to conduct a study that investigates how much both, those static analyses and the mechanics for efficient monitor code generation can be generalized. In particular, one has to answer the question of *exactly what information needs to be known about a given specification formalism or the given specification itself in order to make code generation and analysis feasible*. Once this study has been conducted, the implementation of both, code generation and static analysis, will be generalized accordingly.

Easier specification formalisms: In a fourth and final step we will then concentrate on specification languages. We plan to study existing, potentially graphical, notations in detail. For each such notation we wish to answer the question of *why it is easier or harder to understand than others*. Some particular formalisms could also only be suited for special kinds of specifications. In that case we wish to determine the essence of such specifications and *why they are harder to express in other formalisms*. Finally, we plan to provide a prototypical front-end that allows users to denote specifications in different formalisms which seem particularly suited for large-scale mainstream software development. The addition of another domain-specific language could help answer the question of *whether or not domain-specificity in this setting can make sense*.

3.1 Evaluation

Well-known benchmarks exist for the evaluation of runtime overheads and the precision of static analysis. In our work to date we made use of the DaCapo benchmark suite which consists of ten medium-sized to large-scale Java applications. We plan to conduct consistent experiments with this and other benchmark suites throughout the entire project. One major contribution of our work will be to provide a set of specifications that apply to those benchmarks along with a detailed account of how the various optimizations behave on those specifications and which of the specifications are actually violated by the programs.

With respect to specification formalisms, the question of how those could be evaluated best, remains still unclear at the current time. Even if one had access to subjects willing to try various formalisms and compare them on a subjective basis, it would be hard to guarantee internal and external validity due to potentially different background knowledge of the subjects and due to the large variety of formalisms to choose from. In general, we tend to believe that graphical notations could improve comprehensibility a lot. Yet, this might be hard to prove. Hence, we would be very grateful for comments on that matter.

References

1. *1st to 7th Workshop on Runtime Verification (RV'01 - RV'07)*, 2007. <http://www.runtime-verification.org/>.
2. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications*, pages 345–364. ACM Press, 2005.
3. AspectJ Eclipse Home. The AspectJ home page. <http://eclipse.org/aspectj/>, 2003.
4. P. Avgustinov, J. Tibble, E. Bodden, O. Lhoták, L. Hendren, O. de Moor, N. Ongkingco, and G. Sittampalam. Efficient trace monitoring. Technical Report abc-2006-1, <http://www.aspectbench.org/>, 03 2006.
5. E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object-Oriented Programming*. Springer, July 2007. To appear in *Lecture Notes of Computer Science*.
6. G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28A(4), 1996.

Flexible Ownership Domain Annotations for Expressing and Visualizing Design Intent

Marwan Abi-Antoun and Jonathan Aldrich (Advisor)

School of Computer Science
Carnegie Mellon University

{marwan.abi-antoun, jonathan.aldrich}@cs.cmu.edu

Abstract. Flexible ownership domain annotations can express and enforce design intent related to encapsulation and communication in real-world object-oriented programs.

Ownership domain annotations also provide an intuitive and appealing mechanism to obtain a sound visualization of a system’s execution structure at compile time. The visualization provides design intent, is hierarchical, and thus more scalable than existing approaches that produce mostly non-hierarchical raw object graphs.

The research proposal is to make the ownership domains type system more flexible and develop the theory and the tools to produce a sound visualization of the execution structure from an annotated program and infer many of these annotations semi-automatically at compile time.

1 Problem Description

To correctly modify an object-oriented program, a developer often needs to understand both the code structure (static hierarchies of classes) and the execution structure (dynamic networks of communicating objects). Several tools can extract class diagrams of the static structure from code. To address the problem of extracting the execution structure of an object-oriented program, several static and dynamic analyses have been proposed.

Existing dynamic analyses, e.g., [1, 2] suffer from several problems. First, runtime heap information does not convey design intent. Second, a dynamic analysis may not be repeatable, i.e., changing the inputs or exercising different use cases might produce different results. Third, a dynamic analysis cannot be used on an incomplete program, e.g., to analyze a framework separately from its instantiation. Finally, some dynamic analyses carry a significant runtime overhead — a 10X-50X slowdown in one case [2], which must be incurred each time the analysis is run. Existing compile-time approaches visualize the execution structure using heavyweight and thus unscalable analyses [3] or produce non-hierarchical views that do not scale or provide design intent [3, 4].

Example: JHotDraw. JHotDraw [5] has 15,000 lines of Java code rich with design patterns. However, existing compile-time tools that extract class diagrams or raw object graphs of the execution structure from the implementation do not convey its Model-View-Controller (MVC) design [4] (See Figure 1(a)).

2 Thesis Statement

Hypothesis #1: Flexible ownership domain annotations can express and enforce design intent related to encapsulation and communication in real object-oriented programs. Ownership domains [6] divide objects into *domains* — i.e., conceptual groups, with explicit policies that govern references between

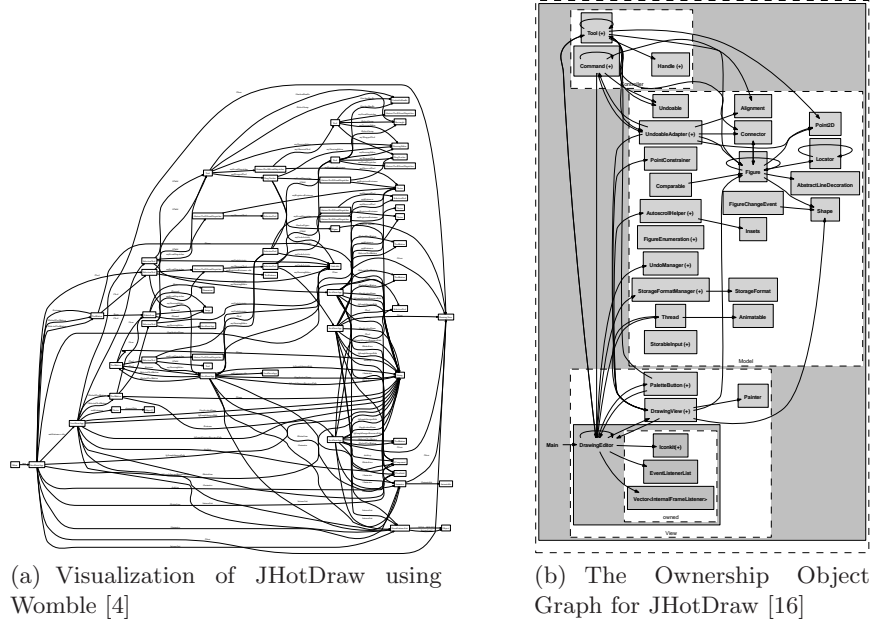


Fig. 1. Side-by-side comparison of compile-time visualizations.

ownership domains. Each object is in a single ownership domain and each object can in turn declare one or more public or private domains to hold its internal objects, thus supporting hierarchy. An ownership *domain* can convey design intent and represent an architectural tier — e.g., the **Model** tier in the MVC pattern [7] — and a *domain link* can abstract permissions of when objects in two domains are allowed to communicate [6] — e.g., objects in the **View** domain can communicate with objects in the **Model** domain, but not vice versa.

Preliminary Work. We added ownership domain annotations to two real 15,000-line Java programs, one developed by experts and one by novices [7]. In the process, we encountered expressiveness challenges in the type system that we plan on addressing. For instance, due to *single ownership*, an object can be in only one ownership domain which makes it difficult to annotate listeners [7] and existential ownership [8, 9] may increase the expressiveness [7].

Expected Contribution #1: We plan to make the ownership domains type system by Aldrich et al. more expressive and more flexible. We will evaluate the modified set of annotations using case studies on non-trivial programs.

Hypothesis #2: Imposing an ownership hierarchy on a program’s runtime structure through ownership domain annotations provides an intuitive and appealing mechanism to obtain a sound visualization of a system’s execution structure at compile time. Furthermore, the visualization is hierarchical and provides design intent.

By grouping objects into clusters called *domains*, ownership domain annotations provide a coarse-grained *abstraction* of the structure of an application — an important goal for a visualization [10, 11] and for scaling to larger programs.

Since the ownership domains type system guarantees that two objects in two different domains cannot be aliased, the analysis can distinguish between instances of the same class in different domains, which would be merged in a class diagram. This produces more precision than aliasing-unaware analyses [4] and more scalability than more precise but more heavyweight alias analyses [3].

Moreover, ownership domain names are specified by a developer, so they can convey abstract design intent more than arbitrary aliasing information obtained using

a static analysis that does not rely on annotations [12]. Finally, unlike approaches that require annotations just to obtain a visualization [13], ownership annotations are useful in their own right to enforce object encapsulation as illustrated by the existing research into ownership types [14, 8, 6, 15].

The novel contribution is the idea of visualizing the execution structure based on ownership domains. Compared with compile-time visualizations of code structure, ownership domains allow a visualization to include important information about the program’s runtime object structures. Compared with dynamic ownership visualizations – which are descriptive and show the ownership structure in a single run of a program, a compile-time visualization is prescriptive and shows ownership relations that will be invariant over all program runs. Thus, this class of visualization is new and valuable.

Preliminary Work. We defined a visualization of the execution structure based on ownership domain annotations, the Ownership Object Graph [16]. We evaluated the visualization on two real 15,000-line Java programs that we previously annotated. In both cases, the automatically generated visualizations fit on one page, illustrated the design intent — e.g., JHotDraw’s MVC design [16] — and gave us insights into possible design problems. The visualization is complementary to existing visualizations of the code structure and compares favorably with flat object graphs that do not fit on one readable page [16] (See Figure 1(b)).

Expected Contribution #2: The Ownership Object Graph would be most useful if it were *sound*, i.e., it should not fail to reveal relationships that may actually exist at runtime, up to a minimal set of assumptions regarding reflective code, unannotated external libraries, etc. Otherwise, the technique would not be an improvement over existing unsound heuristic approaches that do not require annotations and assume that the graph can be post-processed manually to become readable [4]. We plan to augment our formal definition of the Ownership Object Graph with a formal proof of soundness by defining the invariants imposed on the generated data structures, their well-formedness rules and relate the visualization objects abstractly to the program’s runtime object graph.

Hypothesis #3: Once a developer initially adds a small number of annotations manually, it is possible to infer a large number of the remaining annotations semi-automatically. The visualization requires ownership domain annotations, but adding these annotations manually to a large code base is a significant burden. It is precisely such large systems where meaningful views of the execution structure would be most beneficial. In our experience, simple defaults can only produce between 30% and 40% of the annotations [7]. We plan to extend the earlier work on compile-time annotation inference by Aldrich et al. [17] and improve its precision and usability. We plan to develop an approach whereby a developer indicates the design intent by providing a small number of annotations manually; scalable algorithms and tools then infer the remaining annotations automatically. Finally, the Ownership Object Graph helps the developer visualize and confirm the source code annotations.

Expected Contribution #3: We will develop a semi-automated interactive inference tool to help a developer add annotations to a code base without running the program. We will evaluate the tool by taking the programs that we previously annotated manually, removing the annotations and then using the tool to infer the annotations for that code. We chose this methodology since adding ownership domain annotations is often accompanied by refactoring to reduce coupling, to program to an interface instead of a class or to encapsulate fields [7]. Finally, we will compare the visualization obtained from the manually annotated program to the one obtained from the program with the tool-generated annotations.

3 Conclusion

The thesis of this research proposal revolves around adding ownership domain annotations to a program because: a) the annotations can be flexible yet express and enforce the design intent directly in code; b) the annotations can help produce a sound visualization of the execution structure which complements existing visualizations of the code structure; and c) many of these annotations can be inferred semi-automatically.

The goal is to make the ownership domains type system flexible and expressive enough to handle real-world complex object-oriented code. The research will then produce the theory and the tools to obtain a sound visualization of the execution structure from an annotated program and infer many of these annotations semi-automatically at compile time. The type system, visualization and inference will be evaluated in several case studies on real code.

Acknowledgements

This work was supported in part by NSF grant CCF-0546550, DARPA contract HR00110710019, the Department of Defense, and the Software Industry Center at CMU and its sponsors, especially the Alfred P. Sloan Foundation.

References

1. Rayside, D., Mendel, L., Jackson, D.: A Dynamic Analysis for Revealing Object Ownership and Sharing. In: Workshop on Dynamic Analysis. (2006) 57–64
2. Flanagan, C., Freund, S.N.: Dynamic Architecture Extraction. In: Workshop on Formal Approaches to Testing and Runtime Verification. (2006)
3. O’Callahan, R.W.: Generalized Aliasing as a Basis for Program Analysis Tools. PhD thesis, Carnegie Mellon University (2001)
4. Jackson, D., Waingold, A.: Lightweight Extraction of Object Models from Bytecode. *IEEE Transactions on Software Engineering* **27** (2001) 156–169
5. Gamma, E. et al.: JHotDraw. <http://www.jhotdraw.org/> (1996)
6. Aldrich, J., Chambers, C.: Ownership Domains: Separating Aliasing Policy from Mechanism. In: ECOOP. (2004) 1–25
7. Abi-Antoun, M., Aldrich, J.: Ownership Domains in the Real World. In: Intl. Workshop on Aliasing, Confinement and Ownership. (2007)
8. Clarke, D.: Object Ownership & Containment. PhD thesis, University of New South Wales (2001)
9. Lu, Y., Potter, J.: Protecting Representation With Effect Encapsulation. In: POPL. (2006) 359–371
10. Sefika, M., Sane, A., Campbell, R.H.: Architecture-Oriented Visualization. In: OOPSLA. (1996) 389–405
11. Lange, D.B., Nakamura, Y.: Interactive Visualization of Design Patterns Can Help in Framework Understanding. In: OOPSLA. (1995) 342–357
12. Rayside, D., Mendel, L., Seater, R., Jackson, D.: An Analysis and Visualization for Revealing Object Sharing. In: Eclipse Technology eXchange (ETX). (2005) 11–15
13. Lam, P., Rinard, M.: A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In: ECOOP. (2003) 275–302
14. Noble, J., Vitek, J., Potter, J.: Flexible Alias Protection. In: ECOOP. (1998)
15. Dietl, W., Müller, P.: Universes: Lightweight Ownership for JML. *Journal of Object Technology* **4** (2005) 5–32
16. Abi-Antoun, M., Aldrich, J.: Compile-Time Views of Execution Structure Based on Ownership. In: Intl. Workshop on Aliasing, Confinement and Ownership. (2007)
17. Aldrich, J., Kostadinov, V., Chambers, C.: Alias Annotations for Program Understanding. In: OOPSLA. (2002) 311 – 330

Modelling Change-based Software Evolution

Romain Robbes *and* Michele Lanza (Advisor)

Faculty of Informatics
University of Lugano, Switzerland

Abstract. More than 90% of the cost of software is due to maintenance and evolution¹. We claim that the commonly held vision of a software as a set of files, and its history as a set of versions does not accurately represent the phenomenon of software evolution: Software development is an incremental process more complex than simply writing lines of text. To better understand and address the problem of software evolution we propose a model of software in which *change* is explicitly represented to closely match how object-oriented software is implemented. To validate our approach we implemented this model in an Integrated Development Environment (IDE) and used it to perform software evolution analysis. Further validation will continue with the implementation of tools exploiting this change-based model of software to assist software development.

1 Introduction and Problem Statement

Once implemented, a software system has to adapt to new requirements to stay useful [1]: Over time, systems grow and become more complex. Maintaining these systems is hard since developers deal with a large code base they might not fully understand when performing modifications. They have to identify where in the system to apply the change, and keep track of numerous parameters to not introduce bugs during these interventions. 40 % of bugs are indeed introduced while correcting previous bugs [2].

Several approaches exist to assist software evolution. Agile methodologies [3] acknowledge that change is inevitable, rather than attempting to prevent it, and hence ensure that a system is as simple and easy to change as possible. Refactorings [4] are program transformations improving the structure of code, without modifying its behavior, making it easier to maintain. The research field of Software Configuration Management (SCM) [5] built tools to ease versioning, configuring and building large software systems. Finally, reverse engineering environments [6] use the structure and the history of a system to ease a subsequent reengineering effort, the history being often extracted from a versioning system. Indeed, the history of a system contains valuable information [7], [8].

We approach the problem of software evolution at a more fundamental level. Our thesis is that *an explicit model of software evolution improves the state of the art in software engineering and evolution*. Such a model must closely reflect

¹ <http://www.cs.jyu.fi/~koskinen/smcosts.htm>

the reality of software development, which is very incremental in nature. Hence, our model describes the phenomenon of change itself with great accuracy, *i.e. treats change to the system as a first-class entity*. In the following we describe the model we defined, then present our ongoing validation of it.

2 Our approach

Our thesis is that a change-based model of software evolution is beneficial to software development and maintenance activities. By change-based, we mean that the incremental development activities of maintainers and developers must be modelled as changes to closely reflect what happens.

The model of the evolution of software systems we defined follows these principles (more detail can be found in [9]):

Program Representation: We represent a state of a program as an abstract syntax tree (AST) of its source code. We are hence close to the actual system, at the price of being language-dependent. In our model, a program state (of an object-oriented program), contains packages, classes, methods, and program statements. Each program entity is a node of the tree with a parent, 0 or more children, and a set of properties (such as name, type of the entity, comments, *etc.*).

Change Representation: We represent changes to the program as explicit change operations to its abstract syntax tree. A change operation is executable, and when executed takes as input a program state and returns an altered program state. Since each state is an AST, change operations are tree operations, such as addition or removal of nodes, and modifications of the properties of a node.

Change Composition: Low-level change operations can be composed to form higher-level change operations typically performed by developers. For example, the addition of a method is the addition of the method itself, as well as the statements defined into it. At a higher level, refactorings are composed of several “developer-level actions” (*i.e.* renaming a method actually modifies the renamed method and all methods calling it). At an even higher level, our model includes development sessions, which regroup all the changes performed by a developer during a given period of time.

Change Retrieval: This model of the change-based evolution of programs is extracted from IDE interactions of programmers, and stored into a change-based repository. Advanced IDEs such as Eclipse, VisualWorks or Squeak contain enough information and the necessary infrastructure to gather the evolutionary data. This repository is then accessible to tools which can use it to provide useful information to programmer or help him or her performing change requests.

3 Validation

To validate our ideas, we are employing the following 4-step approach:

1. Comparison to other models of evolution: We first surveyed existing models of evolving software used by evolution researchers. These models are closely based on the underlying model of versioning systems, hence we reviewed those in [10]. We concluded that versioning systems used by researchers (because of their popularity among developers, such as CVS or SubVersion) all have similar characteristics: They version systems at the file level to be language-independent, and take snapshots of the system at the developer's request. [10] also outlines why these two characteristics make them not accurate enough to perform finer-grained evolution research.

2. Feasibility: We implemented our model and populated it by developing an IDE plug-in which listens to programmer interactions in the IDE and stores the changes corresponding to these interactions. This IDE plug-in was implemented for the Squeak Smalltalk IDE. It has been ported to VisualWorks Smalltalk, and is being ported to Eclipse for the Java language.

3. Analysing evolution: We then used the data gathered in our repository to perform software evolution analysis, in order to improve program comprehension and reverse engineering. We implemented several visualization and exploration tools on top of the repository. These tools show promising improvements with respect to traditional software evolution analysis based on versioning system data. The information we gather is more precise. Since each change is analysed in context, origin analysis [11] is simplified. We can record precise time information for each change, and reconstitute accurate development sessions [12], whereas traditional approaches can only recover the end result of a session. Analysing such precise sequences allowed us to define new metrics on the sequence of changes itself, measuring for example the activity (number of changes per hour), or the number of changes per entity.

4. Assisting Evolution: If we can record changes to a system, it is also possible to generate these changes. To validate our model in a forward engineering context, we will implement tools designed to ease changing the software itself, using our change-based representation. Several tools can be implemented to validate our approach. One obvious possibility is the implementation of a language-level undo system. We also think a change-based representation could act as a common platform between refactoring tools and other program transformation tools [13]. Code clones could have changes to one instance applied to other instances, in the spirit of [14].

4 Conclusion

To reflect on the phenomenon of software evolution more accurately, we introduced a model of the evolution of programs, in which changes are first-class entities. We represent programs as ASTs and their history as change operations on the AST. These change operations can be composed to represent higher-level

changes such as refactorings, or development sessions. Our ongoing validation shows that our approach recovers more information than found in classical versioning system repositories. However, our approach has several drawbacks: it is language-dependent and requires the presence of an IDE to be accurate. To pursue our validation we are currently porting the approach from Smalltalk to Java, in order to isolate the language-independent parts of our model. We are also building tools exploiting our change-based model to assist software evolution rather than analysing it.

References

1. Lehman, M., Belady, L.: Program Evolution: Processes of Software Change. London Academic Press, London (1985)
2. Purushothaman, R., Perry, D.E.: Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering* **31** (2005) 511–526
3. Beck, K.: Extreme Programming Explained: Embrace Change. Addison Wesley (2000)
4. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison Wesley (1999)
5. Estublier, J., Leblang, D., van der Hoek, A., Conradi, R., Clemm, G., Tichy, W., Wiborg-Weber, D.: Impact of software engineering research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology* **14** (2005) 383–430
6. Nierstrasz, O., Ducasse, S., Girba, T.: The story of Moose: an agile reengineering environment. In: *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, New York NY, ACM Press (2005) 1–10 Invited paper.
7. Girba, T., Lanza, M., Ducasse, S.: Characterizing the evolution of class hierarchies. In: *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, Los Alamitos CA, IEEE Computer Society (2005) 2–11
8. Zimmermann, T., Weißgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. In: *26th International Conference on Software Engineering (ICSE 2004)*, Los Alamitos CA, IEEE Computer Society Press (2004) 563–572
9. Robbes, R., Lanza, M.: A change-based approach to software evolution. In: *ENTCS* volume 166, issue 1. (2007) 93–109
10. Robbes, R., Lanza, M.: Versioning systems for evolution research. In: *Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution)*, IEEE Computer Society (2005) 155–164
11. Tu, Q., Godfrey, M.W.: An integrated approach for studying architectural evolution. In: *10th International Workshop on Program Comprehension (IWPC'02)*, IEEE Computer Society Press (2002) 127–136
12. Robbes, R., Lanza, M.: Characterizing and understanding development sessions. In: *Proceedings of ICPC 2007*. (2007) to appear
13. Robbes, R., Lanza, M.: The “extract refactoring” refactoring. In: *ECOOP 2007 Workshop on Refactoring Tools*. (2007) to appear
14. Duala-Ekoko, E., Robillard, M.P.: Tracking code clones in evolving software. In: *ICSE*. (2007) 158–167

A Rewriting Approach to the Design and Evolution of Object-Oriented Languages

Mark Hills and Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign, USA
201 N Goodwin Ave, Urbana, IL 61801
{mhills,grosu}@cs.uiuc.edu
<http://fsl.cs.uiuc.edu>

Abstract. Object-oriented language concepts have been highly successful, resulting in a large number of object-oriented languages and language extensions. Unfortunately, formal methods for defining and reasoning about these languages are still often performed after the fact, potentially resulting in ambiguous, overly complex, or poorly understood language features. We believe it is important to bring the use of formal techniques forward in this process, using them as an aid to language design and evolution. To this end, we propose a set of tools and techniques, making use of rewriting logic, to provide an interactive environment for the design and evolution of object-oriented languages, while also providing a solid mathematical foundation for language analysis and verification.

Key words: object-oriented languages, programming language semantics, language design, rewriting logic, formal analysis

1 Problem Description

Object-oriented languages and design techniques have been highly successful, with OO languages now used for many important applications in academia and industry. Along with commonly-used static languages, such as Java and C++, there has been a resurgence in the use of dynamic languages, such as Python, and domain-specific languages, often built on top of existing OO languages. This has led to a flurry of research activity related both to the design and formal definition of object-oriented languages and to methods of testing and analyzing programs.

Unfortunately, even as object-oriented languages are used in more and more critical applications, formal techniques for understanding these languages are still often post-hoc attempts to provide some formal meaning to already existing language implementations. This decoupling of language design from language semantics risks allowing features which seem straight-forward on paper, but are actually ambiguous or complex in practice, into the language. Some practical examples of this arose in the various designs of generics in Java, where interactions between the generics mechanism, the type system, and the package-based visibility system led to some subtle errors[2]; other ambiguities in Java have also

Mark Hills, Grigore Roşu

been documented [3]. Decoupling also makes analysis more difficult, since the meaning of the language often becomes defined by either a large, potentially ambiguous written definition or an implementation, which may be a black box.

With this in mind, it seems highly desirable to provide support for formal definitions of languages during the process of language design and evolution. However, existing formal tools are often not suitable to defining the entirety of a complex language, which leads directly to their delayed use. Language definitions based around structural operational semantics [21] and natural semantics [14] both lead to natural methods for executing programs and creating supporting tools [1], but both face limitations in supporting complex control flow features, modularity, and (with natural semantics) concurrency [19]. Modular Structural Operational Semantics [20] is an attempt to improve modularity of definitions, but is still limited by difficulty in supporting some control flow constructs. Reduction semantics [8] provides strong support for complex control flow, but currently has limited tool support with little focus on language analysis beyond typing. Denotational methods have been created to improve modularity [18], but can be quite complex, especially when dealing with concurrency. Current rewriting-based methods provide powerful tool support [23, 17] and can be conceptually simpler, but can be verbose and non-modular. A formal framework, concise, modular, and broadly usable, providing support for defining even complex language features, while providing tools for language interpretation and analysis, is thus critical to opening up the use of formal techniques during language design.

2 Goal Statement

The goal of our research is to provide an environment for the design and evolution of programming languages that is based on a solid formal foundation. This environment should be flexible enough to define the complexities of real languages, such as Smalltalk, Java, Python, or Beta, with support for the rapid prototyping of new languages and language features. Definitions should be formal, executable, and modular, allowing the user to define new features and immediately test them on actual programs. Analysis should also be easily supported, providing the ability to check existing programs and to ensure that new language features (especially those related to areas like concurrency, which can have unexpected interactions with other language features) work as expected.

Overall, we expect our research to produce: a framework for the modular definition of languages, including a number of pre-defined language modules; definitions of multiple new and existing languages, available as the basis for language extensions and as example definitions; graphical tools for joining together language modules and animating the execution of the semantics (actual program execution, typing, abstract interpretation, etc); and translations into various runtime environments for the execution and analysis of programs using the defined language semantics. We believe this framework, with the associated tools and definitions, can help make formal techniques useful not only for post-hoc studies of languages, but also during the language design process, providing formal definitions as a natural part of language design and evolution.

3 Approach

A Framework for Language Definition: Our work on programming languages has focused mainly on the use of rewriting logic [16, 15], a logic of computation that supports concurrency. While we believe rewriting logic, in combination with engines such as Maude [5, 6], is a compelling solution for defining and reasoning about languages [17], the generality of rewriting logic can sometimes lead to verbose, non-modular language definitions [22]. To exploit the strengths of rewriting logic, while ameliorating some of the weaknesses, we have begun work on K, a domain-specific variant of rewriting logic focused on defining programming languages [22]. K is specifically being designed to provide for concise, modular definitions of languages, written in an intuitive style. Initial versions of K have been used in the classroom and to define portions of Java and an experimental object-oriented language named KOOL [4, 11].

Defining and Evolving Object-Oriented Languages: An important test of our techniques is to define and experiment with actual object-oriented languages. This has been done both using K and directly in rewriting logic. One result has been the KOOL language, a dynamic, concurrent, object-oriented language [12], designed specifically to experiment with a variety of language extensions. Other work has resulted in an initial rewriting logic definition of Beta [9] and definitions of Java [7] and JVM bytecode [7]. We plan to extend this work, not only defining languages but also building libraries of language features, while using the feedback from this process to improve the K framework.

Tool Support: To make the K framework and associated language definition techniques widely useful, we plan to develop tools to support language design, analysis, and execution. We have done some initial work on how language design decisions impact analysis performance [13], and our work on KOOL [12] provided an interesting example of how analysis can be used to check that language features work as expected. Other initial work has demonstrated the promise of translating language definitions in K into executable form [10]. Additional work will involve developing a user interface for working with K definitions and animation tools to visualize the workings of the language semantics, with further work on translations directly from K to Maude and various target languages.

References

1. P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of SDE 3*, pages 14–24. ACM Press, 1988.
2. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *OOPSLA '98*, pages 183–200, New York, NY, USA, 1998. ACM Press.
3. J.-T. Chan, W. Yang, and J.-W. Huang. Traps in Java. *J. Syst. Softw.*, 72(1):33–47, 2004.

Mark Hills, Grigore Roşu

4. F. Chen, M. Hills, and G. Roşu. A Rewrite Logic Approach to Semantic Definition, Design and Analysis of Object-Oriented Languages. Technical Report UIUCDCS-R-2006-2702, University of Illinois at Urbana-Champaign, 2006.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In *Proceedings of RTA'03*, volume 2706 of *LNCS*, pages 76–87. Springer, 2003.
7. A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal Analysis of Java Programs in JavaFAN. In *Proceedings of CAV'04*, volume 3114 of *LNCS*, pages 501–505. Springer, 2004.
8. M. Felleisen and R. Hieb. A Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 103(2):235–271, 1992.
9. M. Hills, T. B. Aktemur, and G. Roşu. An Executable Semantic Definition of the Beta Language using Rewriting Logic. Technical Report UIUCDCS-R-2005-2650, University of Illinois at Urbana-Champaign, 2005.
10. M. Hills, T. F. Şerbănuţă, and G. Roşu. A rewrite framework for language definitions and for generation of efficient interpreters. In *Proceedings of WRLA'06*, ENTCS. Elsevier, 2007. To appear.
11. M. Hills and G. Roşu. KOOL: A K-based Object-Oriented Language. Technical Report UIUCDCS-R-2006-2779, University of Illinois at Urbana-Champaign, 2006.
12. M. Hills and G. Roşu. KOOL: An Application of Rewriting Logic to Language Prototyping and Analysis. In *Proceedings of RTA'07*, *LNCS*. Springer, 2007. To appear.
13. M. Hills and G. Roşu. On Formal Analysis of OO Languages using Rewriting Logic: Designing for Performance. In *Proceedings of FMOODS'07*, volume 4468 of *LNCS*, pages 107–121. Springer, 2007.
14. G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences*, *LNCS*, pages 22–39. Springer, 1987.
15. N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.
16. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
17. J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, to appear, 2007.
18. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh, 1989.
19. P. D. Mosses. The varieties of programming language semantics. In D. Björner, M. Broy, and A. V. Zamulin, editors, *Ershov Memorial Conference*, volume 2244 of *LNCS*, pages 165–190. Springer, 2001.
20. P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, July-December 2004 2004.
21. G. D. Plotkin. Lecture notes DAIMI FN-19: A Structural Approach to Operational Semantics. Dept. of Computer Science, University of Aarhus, 1981.
22. G. Roşu. K: a Rewrite Logic Framework for Language Design, Semantics, Analysis and Implementation. Technical Report UIUCDCS-R-2006-2802, University of Illinois at Urbana-Champaign, 2006.
23. M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM TOPLAS*, 24(4):334–368, 2002.



PhD Students Workshop

Checking Semantic Usage of Frameworks

Ciera Christopher Jaspan

Institute for Software Research
Carnegie Mellon University
Pittsburgh PA 15213, USA
`cchristo@cs.cmu.edu`

Abstract. Software frameworks are difficult for plugin developers to use, even when they are well designed and documented. Many of the framework problems stem from *constraint inversion* and the inability for users to know all possible constraints that they may be breaking in the framework. These constraints are relative to the context of the plugin, and they can involve multiple framework objects. This paper describes the beginnings of a language to check framework usage from a semantic perspective, rather than a purely structural view.

1 Software Frameworks

Software frameworks allow developers to reuse not just an implementation, but a complete architecture and design. By abstracting and reusing the architecture, the programmer also receives the benefit of architectural solutions to difficult issues such as scalability, concurrency, security, and performance. However, there is a cost to using a framework; software frameworks are very complex and difficult to learn. [1]

Software frameworks depend heavily on the notion of a *callback* method. These methods are defined by the framework through interfaces, but they are implemented by *plugin* code in the style of the Template design pattern. [2] The plugin provides customized functionality to the framework, and the result of combining the framework and plugin is a complete application. .

This notion of *callback* is the major difference between frameworks and libraries. In essence, it is describing who has control of the program flow. In a library, the user of the library, or *application developer*, is responsible for the architecture and control flow. This relationship is inverted in a framework. A framework is in charge of the major control flow, and the application developer can only control their customized plugin code.

This inversion of control also causes *constraint inversion*. A library certainly has internal constraints, but they are encapsulated and hidden from the application developer. The constraints which must be seen by the application developer are known as the library's *protocol*. Enforcement of protocols is ongoing but mature research.[3–5] In a framework, these roles are reversed. The framework defines a protocol for how it will call the application code; this is called a *lifecycle*. However, the framework must also expose its own constraints since the plugin code needs direct access to many internals. It is very easy for an application developer to unknowingly break a constraint because there is no enforcement of what they can and cannot do within their callback methods, even though the framework is expecting the plugin to meet certain conditions.

It is very difficult for even experienced developers to keep track of all the constraints that they must comply with. We propose to ease this burden by providing language support to discover mismatches between the plugin code and the declared constraints of the framework. Our solution is guided by the following principles:

1. *No effort for the plugin developer* The plugin developer should not have to make any additional effort to use the framework. In particular, they do not add any specifications to their plugin code.
2. *Minimal effort for the framework developer* The framework developer will have to specify how to use the framework and what constraints exist. This should not require a complete specification of the framework's internals.
3. *Localized errors* Frameworks require developers to use many objects from different places in the framework. The constraints are inherently distributed across all of these objects. In order for this solution to be adoptable, errors from incorrect usage can not simply state that there was an error in a constraint; errors should be specific to the plugin code and should give a localized description.

To better understand framework constraints, we will explore two examples from the ASP.NET framework. We will use these examples to motivate *relationships* and *scopes* as a way to discover semantic defects in framework usage.

2 Motivating Examples

To motivate this work, we will examine two examples from the ASP.NET framework. The first example, DropDownList Selection, is from the author's own experiences with ASP.NET. The other example, Login Status, was mined from the ASP.NET help forums. [6]

2.1 DropDownList Selection

The ASP.NET framework allows developers to create pages with web controls on them. These controls can be manipulated programatically through the callbacks provided by the framework. Developers can respond to control events, add and remove controls, and change their state.

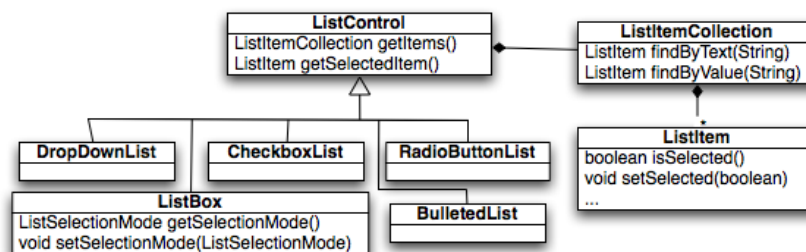


Fig. 1. ListControl Class Diagram

One task that a developer might want to do is programmatically change the selection of a drop down list. The ASP.NET framework provides us with the relevant pieces as shown in Figure 1.¹ Notice that if we want to change the selection of a **DropDownList** (or any other derived **ListControl**), we have to access the individual **ListItems** through the **ListItemCollection** and change the selection there. Based on this information, a developer might naively change the selection as shown in Program 1.

Program 1 Incorrectly selecting multiple items in a DropDownList

```

DropDownList list;

private void Page_Load(object sender, EventArgs e) {
    string searchVal = ...
    ListItem item = list.getItems().findByValue(searchVal);
    item.setSelected(true);
}

```

Program 2 Correctly selecting an item using the API

```

DropDownList list;

private void Page_Load(object sender, EventArgs e) {
    string searchVal = ...
    list.getSelectedItem().setSelected(false);
    ListItem item = list.getItems().findByValue(searchVal);
    item.setSelected(true);
}

```

This code breaks an important framework constraint. A `DropDownList` must have one item selected, and it may only have one item selected. Selecting multiple items causes a runtime exception when the `DropDownList` is rendered, while selecting no items causes the framework to select the first item in the list. To make this problem more interesting, the selection constraint changes for each derived class of `ListControl`, even though `ListControl` defines selection mechanism. For example, a `CheckBoxList` may have 0-n items selected, and a `ListBox`'s selection constraint changes according to a `SelectionMode` property.

Program 2 shows a correct example for this task. By the time we end the method, there is only one item selected.

2.2 Login Status

On the ASP.NET forums, a developer reported that he was attempting to retrieve a `DropDownList` from his page, but his code was throwing a `NullReferenceException`. [7] An abbreviated version of the ASPX file is in Program 3, and the code snippet causing the error is in Program 4. Another developer responded to the post and explained the issue. The `LoginView` will only contain the controls in the `LoggedInTemplate` if the user is authenticated. If not, those controls will not even exist. The solution proposed was to first check the log in status from `Request.IsAuthenticated`, as shown in the corrected Program 5.

In both of these examples, the plugin developer broke an unknown framework constraint. They had used the framework in a way which seemed intuitive because they had a slightly misshapen view of what the framework was doing. We can not realistically expect that a plugin developer will understand all of the internals of a framework, so we propose a framework specification that will discover these issues without input from the plugin developer.

¹ To make this code more accessible to those unfamiliar with C#, I am using traditional getter/setter syntax rather than properties.

Program 3 ASPX with a LoginView

```

<asp:LoginView ID="LoginScreen" runat="server">
  <AnonymousTemplate>
    You can only set up your account when you are logged in.
  </AnonymousTemplate>
  <LoggedInTemplate>
    <h2>Select a Membership</h2>
    <asp:DropDownList ID="MembershipList" runat="server"/>
    <asp:Button ID="ContinueButton" runat="server" Text="Continue"/>
  </LoggedInTemplate>
</asp:LoginView>

```

Program 4 Incorrect way of retrieving controls in a LoginView

```

private void Page_Load(object sender, EventArgs e) {
  DropDownList list = (DropDownList)LoginScreen.FindControl("MembershipList");
  list.DataSource = Roles.GetAllRoles();
  list.DataBind();
}

```

Program 5 Correct way of retrieving controls in a LoginView

```

private void Page_Load(object sender, EventArgs e) {
  if (Request.IsAuthenticated()) {
    DropDownList list = (DropDownList)LoginScreen.FindControl("MembershipList");
    list.DataSource = Roles.GetAllRoles();
    list.DataBind();
  }
}

```

3 Relationships and Scopes

To express the constraints described above, we use two constructs. The first construct is the *relationships* among objects. Relationships will provide us with a semantic context of the framework. The second construct, *scopes*, allows us to define regions of code in the plugin. Within these regions, new operations may be available, or old operations may be disabled. This section will use the ListControl Selection example to examine how relationships and scopes interact to find defects.

Relationships associate two objects with a user-defined meaning. The attribute `[Child(o1, o2)]` creates a relationship between `o1` and `o2`, while `[Child(o1, o2, false)]` removes this relationship from the current context. Object parameters can be wild-carded, so `[Child(o1, _, false)]` removes all the “Child” relationships from `o1` to any other object. After calling a method, we acquire (or kill) a set of relationships. We track relationships through the plugin code using a dataflow analysis. The current relationships provide us with a context that describes what we know about the framework’s state. The relationship attributes for the ListControl framework are displayed in Program 6.

Program 6 Annotated API of the ListControl

```
public class ListControl {
    [Member(ret, this)]
    ListItemCollection getItems();

    [Child(ret, this)]
    [Selected(ret, true)]
    ListItem getSelectedItem();
}

public class ListControl {
    [Item(ret, this)]
    [Value(val, ret)]
    ListItem findByValue(string val);
}

public class ListItem {
    [Selected(this, true)]
    boolean isSelected();

    [Selected(this, _, false)]
    [Selected(this, select)]
    void setSelected(boolean select);
}
```

Once we can track relationships, we can use *scopes* to describe framework constraints about these relationships. A scope defines a region of code where some operations are allowed or disallowed. In the Login Control example, there is a scope within the if-block that states we can now access the controls within the LoginControl’s `LoggedInTemplate`. These scopes may not be just at regular code block boundaries; as we will see, the `DropDownList` defines a scope that exists between two method calls. We call these places where scopes can start and end *program points*.

Scopes are more than just a syntactic region though. Scopes depend on not only program points, but the relationships that exist at those points in time. By depending on relationships,

we are checking the semantics of the expressions rather than the syntax. This allows us to handle multiple scopes in the code that cover the same objects, or the same scope definitions operating on different objects. A scope is defined by many parts, as described below.

- **declared** These objects are used in the program points and relationships.
- **start** A list of program points and relationships that start the scope. The program points are syntactic expressions while the relationships provide the semantic context that is required. The relationships are a precondition for the scope to start.
- **end** A list of program points and relationships that end the scope. Like **start**, **end** requires relationships; the program point by itself does not end the scope.
- **enable** A list of program points and relationships that are allowed to occur in the scope.
- **disable** A list of program points and relationships that are not allowed to occur in the scope.
- **scoped** A list of objects that are temporary to this scope. They are not valid outside the scope and can not be stored.

We will create a scope for the DropDownList Selection example. For now, we will write this informally, but there is a formal notation for scopes. We create a scope which starts when we deselect the currently selected item on the list, and the scope ends when we select another item. This scope prevents a user from selecting multiple items by forcing the old item to be deselected before selecting a new item. It also prevents a user from leaving nothing selected by disabling the end of the method while we are in the scope.

- **declared:** DropDownList ctrl, ListItem oldSel, ListItem newSel
- **start:** oldSel.SetSelected(false); {Child(oldSel, ctrl), Selected(oldSel, true)}
- **end:** newSel.SetSelected(true); {Child(newSel, ctrl), Selected(oldSel, false)}
- **enable:** newSel.SetSelected(true); {Child(newSel, ctrl), Selected(oldSel, false)}
- **disable:** end of method; {Child(newSel, ctrl), Selected(oldSel, false)}
- **scoped:** none

We will not create the scope for our other example here, but the scope would start and end on the if block. The objects within the `LoggedInTemplate` would only be accessible within that scope.

4 Validation Plan

This work needs to be evaluated in two dimensions: it must be able to succinctly express common framework constraints and it must be able to find defects in plugin code. To evaluate this work, we will run small case studies using real examples in the ASP.NET and Eclipse frameworks. These frameworks each have developer forums where we can find sample problems along with defective plugin code and an explanation of the broken framework constraint. By specifying the framework that this sample code uses, we can show that the specifications are expressive enough to capture the required framework constraint. If time allows, we will also run a larger case study on a single framework. If possible, we will have a framework developer specify their framework, and we will have new plugin developers use the analysis tool over an extended period of time. From these case studies, we would like to show that scopes can specify a wide range of framework constraints and that they are able to find defects early.

5 Future Work

Scopes are the first step toward creating a language to describe framework semantics. In future work, I would like to explore other constructs that will make describing frameworks and checking plugins easier. There are several constructs that frameworks use informally, including lifecycles of plugins, callback methods, and extension points. Future work will explore each of these possible constructs and understand how semantic context affects these constructs.

A framework language would also provide information for new developer tools. For example, a plugin developer might want to know what relationships a particular object is involved in, or what scopes are active at a particular point. If a developer receives an error because they called an operation that was disabled in a current scope, the developer might ask what available operations will close the offending scope. By providing tools with semantic information about the framework and plugin, they can give more directed error reporting and provide more relevant information.

6 Related Work

Some of the original work in frameworks [8] discussed using design patterns as a way of describing frameworks. Later research has looked at formalizing design patterns and extracting design patterns from code [9–11]. Patterns alone can not completely specify a framework. While they provide information about high-level interaction mechanisms, they do not describe the temporal framework constraints shown in our examples.

SCL [12, 13] allows framework developers to create a specification for the structural constraints of using the framework. The proposed framework language focuses on semantic constraints rather than structural constraints. Some of the key ideas from SCL could be used to drive the more structural focused parts of the specifications.

Object typestates [4, 5] provide a mechanism for specifying a protocol between a library and a client. The client sees the library to be in a particular state, and calling methods on the library transitions to a new state. This general concept can also be applied to frameworks and plugins. However, due to inversion of control, the protocol is now on the plugin; in a framework setting, we call this a *lifecycle*. If we continue to use typestates to represent lifecycles, then the plugin methods are the state transitions. This is not how a plugin developer thinks of the code; we would prefer to think of the framework as transitioning the state and the plugin doing specialized code within the current framework state. Additionally, framework states involve multiple interacting objects; this is awkward to model with typestates. A framework language should have a construct to represent lifecycles; while it may be inherently different from a typestate-based protocol, it might be possible to reuse some of the underlying theory.

Some typestate work has explored inter-object typestate. This work still considers each object to have an individual typestate, though it can be affected by other objects [14] or manipulated through participation in data structures [15]. Scopes differ in that they view multiple heterogeneous objects as having a shared state.

Scoped Methods [3] are another mechanism for enforcing protocol. They create a specialized language construct that requires a protocol to be followed within it. Like SCL, this is structural and does not take context into account.

Like the proposed framework language, Contracts [16] also view the relationships between objects as a key factor in specifying systems. A contract also declares the objects involved in the contract, an invariant, and a lifetime where the invariant is guaranteed to hold. Contracts allow all the power of first-order predicate logic and can express very complex invariants. Contracts

differ from the proposed framework language because they do not make the tie directly back to the plugin code and have a higher complexity for the writer of the contract.

Other research projects [17–19] help plugin developers by finding or encoding known good patterns for using frameworks. The proposed work differs significantly in that it does not suggest a way to complete the task, but it finds defects once a task has been started. We see the two bodies of research as complimentary; a framework language should increase the ability to find and encode these patterns by providing a common language.

This work also has some overlap with formal methods, particularly in describing the relationships and invariants of code [20, 21]. These formal methods verify that the specified code is correct to the specification. Instead, we are checking the unspecified plugin code against the framework’s specification.

7 Conclusion

Frameworks place constraints on plugins that are relative to a semantic context of the code. These constraints can be dependent upon many interacting objects, and they can affect the operations which a user has access to.

We plan to create a language that describes the constraints that a framework may place on the plugin. This language would be used to find semantic problems with plugin code, such as how the plugin code uses the framework at points in the lifecycle, which objects the plugin can access and save references to, and what relationships must be maintained between framework objects.

As a first step toward this goal, we have created relationships and scopes to specify framework constraints. Relationships keep track of the knowledge a plugin has based upon its previous framework interactions. Scopes provide a mechanism for describing framework constraints that occur within a region of code. These regions are described by program expressions and relationships so that both the syntactic and semantic nature of the constraint are expressed. An analysis can check that a plugin only uses operations when they are allowed.

In future work, we will explore other framework concepts as additions to the language, such as lifecycles, callbacks, and extension points. Moving these concepts into first class constructs will provide us with a way to check plugins against a framework specification.

8 Acknowledgments

I’d like to thank my advisor, Jonathan Aldrich, for his contributions and guidance with this work. Thanks also to George Fairbanks and Kevin Beirhoff for their insightful discussions. This work was supported in part by NSF grant CCF-0546550, DARPA contract HR00110710019, the Department of Defense, and the Software Industry Center at CMU and its sponsors, especially the Alfred P. Sloan Foundation.

References

1. Johnson, R.E.: Frameworks = (components + patterns). *Commun. ACM* **40**(10) (1997)
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. (1995)
3. Tan, G., Ou, X., Walker, D.: *Enforcing resource usage protocols via scoped methods* (2003) Appeared in the 10th International Workshops on Foundations of Object-Oriented Languages.

4. DeLine, R., Fahndrich, M.: Tpestates for objects. In: ECOOP '04: Proceedings of the 18th European Conference on Object Oriented Programming. (2004)
5. Bierhoff, K., Aldrich, J.: Lightweight object specification with tpestates. In: ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. (2005)
6. : (The asp.net forums) <http://forums.asp.net>.
7. : Binding to a dropdownlist membership roles (2006) <http://forums.asp.net/thread/1415249.aspx>.
8. Johnson, R.E.: Documenting frameworks using patterns. In: OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications. (1992)
9. G. Florijn, M. Meijers, P.v.W.: Tool support for object-oriented patterns. In: ECOOP '97: Proceedings of European Conference of Object-oriented Programming. (1997)
10. D. Heuzeroth, S. Mandel, W.L.: Generating design pattern detectors from pattern specifications. In: 18th IEEE International Conference on Automated Software Engineering. (2003)
11. Soundarajan, N., Hallstrom, J.O.: Responsibilities and rewards: Specifying design patterns. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering. (2004)
12. Hou, D., Hoover, H.J.: Towards specifying constraints for object-oriented frameworks. In: CASCON '01: Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research. (2001)
13. Hou, D., Hoover, H.J.: Using scl to specify and check design intent in source code. *IEEE Transactions on Software Engineering* **32** (2006)
14. Nanda, M.G., Grothoff, C., Chandra, S.: Deriving object tpestates in the presence of inter-object references. In: OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications. (2005)
15. Lam, P., Kuncak, V., Rinard, M.: Generalized tpestate checking for data structure consistency. In: Verification, Model Checking, and Abstract Interpretation. (2005)
16. Helm, R., Holland, I.M., Gangopadhyay, D.: Contracts: specifying behavioral compositions in object-oriented systems. *SIGPLAN Not.* **25**(10) (1990)
17. Froehlich, G., Hoover, H.J., Liu, L., Sorenson, P.: Hooking into object-oriented application frameworks. In: ICSE '97: Proceedings of the 19th international conference on Software engineering. (1997)
18. Mandelin, D., Xu, L., Bod, R., Kimelman, D.: Jungloid mining: helping to navigate the api jungle. In: PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. (2005)
19. Fairbanks, G., Garlan, D., Scherlis, W.: Using framework interfaces with design fragments. In: OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. (2006)
20. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. *SIGPLAN Not.* **37**(5) (2002)
21. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes* **31**(3) (2006)

An Integrated Method based on Multi-Models and Levels of Modeling for Design and Analysis of Complex Engineering Systems

Michel dos Santos Soares and Jos Vrancken

Faculty of Technology, Policy and Management
Delft University of Technology
The Netherlands

m.dossantosoares@tudelft.nl, j.l.m.vrancken@tudelft.nl

Abstract. Model-Driven Systems Engineering (MDSE) is rapidly evolving and new modeling languages, processes, methods, and tools are advancing in parallel to support it. MDSE differs from traditional Systems Engineering, which is characterized as document centric, through the application of models as primary engineering artifacts throughout the engineering life cycle, and dealing with systems decomposition rather than requirements decomposition, increasing possibilities of reusing artifacts. This research is about using and integrating different System and Software Engineering tools and languages to model, design and analyze complex engineering systems. One of the most important features is that complexity is treated in several levels, through abstraction, decomposition and modularization. The main purpose is to create an integrated MDSE method, apply it in the Road Traffic domain and evaluate its advantages, avoiding inconsistency and fragmentation.

1 Introduction

The complexity of large engineering systems makes their development very hard, costly and error-prone. In order to manage this complexity, systems are normally built based on diverse models at different levels of abstraction. Abstraction is a central principle in Software and Systems Engineering, referring mainly to the separation of essential points and leaving what is not essential at a determined stage of the design life cycle.

Models provide abstractions in order to reason about a system. Modeling is an approach to try to understand complex real world constructions and systems through models, which is important to help stakeholders to understand systems and as a communication medium. There's a gap between the needs and constraints, which usually are expressed in natural language and informal diagrams, and the detailed specifications needed to build systems. Modeling can fill this gap, with standard notations that are used and understandable by the stakeholders, improving communication between teams and at least significantly diminishing natural language ambiguities. Besides, models can also be executed

and analyzed. The analysis allows the validation of the system and verification of desirable properties, which contributes to decrease risks and future problems.

One fundamental solution to deal with complex engineering systems is to build models hierarchically. It is not possible to capture functional features and quality properties of complex systems in a single model that is understandable by and of value to all stakeholders [1]. A widely used approach is then to build different but interrelated views, which collectively describes the whole system. Depending on the relevance of a determined stage of product design, different models are constructed using diverse languages/tools in order to center attention on particular perspectives. These diverse views of a system are important when there are several stakeholders involved, with each one interested in more or less details.

The main purpose of model-driven approaches [2] [3] is to build software and systems with a more controlled and efficient process, resulting in higher quality products. This research is about creating a model-driven method to model, design and analyze complex engineering systems with applications to Road Traffic Management Systems. Due to their complexity, the proposed approach is to divide the design of a system into levels. Each level has some goals, and is composed of several models built using specific languages/tools.

2 Problem Description

The term “complex system” is still controversial; the definition depends on the perspectives and objectives of the stakeholders. One definition close to the purpose of this research is: “a system with numerous components and interconnections, interactions or interdependencies that are difficult to describe, understand, predict, manage, design, and/or change” [4]. In [5], the author presents four different views of a complex system: hierarchical mappings, state equations, nonlinear mechanisms and autonomous agents. The chosen view for this research is that of hierarchical mappings, in which complex systems are studied by hierarchical decomposition of a very complicated design into smaller components.

There are several causes of problems that can lead to failures when designing a complex system. For instance, errors in estimating tasks and their durations, a poor interface, and risks not well analyzed. These are important problems, but in this research, software and systems aspects are evaluated instead of economic or psychological ones. A fundamental reason for the difficulties in designing modern large engineering projects is their inherent complexity [6]. Software intensive systems are more complex for their size than perhaps any other human construct [7]. The degree of repetition is low, and every design has some challenges that perhaps have not appeared before.

The lack of communication is a factor that can lead to system failure. In this research, communication is considered in a more broad aspect, as for instance, between subsystems, teams, organizations or stakeholders. Common vocabulary can help in improving the understandability of specifications, as requirements misunderstanding is one principal cause for project failures. In addition, with so

many different methods, languages and formalisms to model, design and analyze systems, it is difficult to decide which approach is more suitable to each problem and part of the life cycle. Incorrect decisions may lead to misuse a formalism or use it in an inappropriate context.

One solution to deal with the problems presented above is to use models to represent the various aspects of systems. Models are useful to specify, visualize, communicate and document systems, and can simplify reality in a domain. Although the application of models for Systems Engineering is not new, systems engineers have significantly increased their use of model-driven development technologies. Traditional approaches, such as requirements-driven systems development methods [8] that apply functional decomposition to map requirements into system components are limited to address the challenges of developing complex systems. As systems become more complex and integrated, this traditional approach becomes difficult to manage due to the large number of possible mappings from the problem to the solution domain [9]. In addition, these methods, based on a stable and predictable set of requirements, are limited in their capability to address models integration, information sharing and requirements changing. Systems engineering methods have to be extended in order to address these problems.

The principal focus of this research is to address the problems presented above applying methods and languages to build models to engineering complex systems. The research domain is in the field of transportation. Transportation systems are complex networked systems that provide accessibility to places and mobility to people and goods [10]. The occurrence of traffic jams is increasing in many countries, which leads to economic losses, augments pollution and has a great impact on the quality of life. One common approach to try to handle traffic congestion is to build more infrastructures, such as roads and bridges. But, in the past years, it is becoming increasingly more difficult to build more infrastructures. Not only the high cost, but also the lack of space and the environmental damage of building new roads have to be considered. A different approach is needed, based on applying intelligence in order to manage traffic flow in a more effective and efficient manner. This leads to a relatively new research area named Intelligent Transportation Systems (ITS) [11]. Basically, ITS is the application of Information and Communication Technologies (ICT) to the planning and operation of transportation systems.

ITS research is multidisciplinary, depending on results from several different research areas, such as electronics, control engineering, communications, sensing, signal processing and information systems [12]. This multidisciplinary nature increases the problem's complexity because it requires knowledge transference, communication and cooperation among diverse research areas and specialists. Difficulties have been addressed through the growing application of Systems Engineering processes and methods for ITS projects and research [13].

Among the several research areas of ITS, one of the main is the management of traffic. Road Traffic Management Systems (RTMS) are complex systems composed of several integrated parts. Some examples of RTMS parts are traf-

fic signals, detectors, dynamic route information panels, intersections and links. The design, modeling and analysis of RTMS can't be done only considering local traffic. For example, it's not feasible to model and analyze separately traffic systems controlling road intersections. To be effective, the network level must be evaluated, and problems solved not only locally, but in a wide area [14]. Otherwise, the risk is not to solve problems, but just move them to another place in the network.

Because of the complexity of RTMS, it is difficult to represent their aspects using a single level, only one method or any single tool/language. Within complex engineering systems, it is necessary to model components with different characteristics and nature, for instance, mechanical, hardware, software, etc. Another difficulty is that initial requirements are normally written in natural language, possibly with aid of informal diagrams, which can be inconsistent and confusing, leading to communication problems with other stakeholders. In addition, informal specifications can't be verified in a formal way in order to guarantee the correct functionality of systems.

3 Related Work

Perhaps the best known model-driven initiative is OMG's Model Driven Architecture (MDA) [15]. A central MDA idea is the separation of specification of the system from the details of its platform. With MDA approach, software development is based on series of model transformation steps, which starts with a high level specification using the domain vocabulary and ends with a platform specific model.

There are several researches on model-driven approaches with application to a variety of complex engineering systems. The aim of project MOVES [16] is to analyze problems raised by the evolution of model-driven approaches. DRIP Catalyst [17] is a framework-specific MDE/MDA method developed combining model-driven, generative and formal techniques to support cost-effective and disciplined development of fault-tolerant applications. The RODIN project [18] aims to contribute to create a methodology and tools, combining UML, Petri nets and the B method, for rigorous development of dependable complex systems. Among the several case studies, one is related to the construction and composition of systems utilizing object-oriented and model-driven techniques for telecommunication systems [19]. In the project "Model-Based System Engineering" [20], the authors use executable specifications, along with advanced analysis tools and simulation environments to evaluate system design before construction begins. Several other researches and information about model-driven techniques can be found at the Planet MDE website ¹.

¹ www.planetmde.org

4 Proposed Approach

A good practice when modeling systems is to use a view of the model that only includes the concepts and relations that are relevant for each stakeholder [21]. Also, maintain traceability between models is an important quality factor. After modeling, for complex engineering systems it is desirable to simulate and formally analyze models to ensure the presence of important properties and absence of problems such as deadlocks. Normally, models are evaluated using inspections and reviews, which are useful to detect simple errors, but are tedious and error-prone when dealing with large models. Executable models, as model-driven approach proposes, are better suited to give feedback to designers of large systems.

Model-Driven System Engineering (MDSE) differs from traditional systems engineering, which is characterized as document centric, through the application of models as primary engineering artifacts throughout engineering life cycle [2]. During system development, models are transformed, and each transformation adds levels of specificity and details. Models can also be refined when more information is added due to better understanding of the system during the life cycle. Although there is already a large body of work reported, a large amount of work remains to be completed for this new discipline to achieve its potential. Because MDSE is still in its infancy, there is a need for formalisms, methods, techniques and associated tools supporting model development, transformation and evolution.

MDSE doesn't deny the success of the requirements-driven approach. For instance, system decomposition into smaller components (divide and conquer) has a historical success. But within MDSE, system decomposition is based on structure rather than by function, enabling the framework to provide several levels of structure. According to [9], MDSE starts with system decomposition into elements, each one with a set of requirements, in order to improve system comprehension and manage complexity. Effective application of system decomposition requires modeling the system from several viewpoints and at increasing levels of detail. The approach is based on a set of transformations between models, with each transformation providing means for deriving the next level of specificity while maintaining traceability with the entire model through the development life cycle.

MDSE consists of creating the models as a mean of specifying the system elements and their integration. System models are constructed normally with formal or semi-formal methods/languages. Semi-formal methods are user oriented but lack mathematical rigor and can't formally prove desirable behavior of systems. Formal methods are well-suitable to analyze system behavior, but have some disadvantages. For instance, their mathematical and logical approach makes it difficult to most of the stakeholders to understand. Another big difficulty is that normally software is developed from informal requirements. So, one problem to be addressed is how to use formal specifications to model informal requirements. And so, once this is done, how to easily change formal specifications every time there's a change in informal requirements by stakeholders. Combining

formal and semi-formal methods is a well-known and applied approach that tries to use the best of each type of method [22].

When comparing the proposed approach of this research and other related works, some similarities are evident, as, for instance, the combination of languages and methods. This is important because of the difficulty of using a single language to several system aspects. On the other hand, the systems view is still not well-considered in earlier projects, and when it is the case, it still lacks a proper systems language. Normally, UML [23] and Petri nets [24] are adapted to model and design systems. In this research, system aspects are taken into account from the early beginning with SysML [25], a specific systems language. Also, in some approaches, the lack of formal knowledge representation is common. To address this problem, Semantic Web technologies, such as ontologies, are applied. The combination of Software Engineering and Knowledge Engineering is a promising research area [26]. Finally, due to the impossibility (and impracticality) of formally analyzing a whole system, normally just some critical and more important parts are analyzed, based on some criteria. The problem here is that the selection of which subsystems/parts deserve special attention still lacks more defined criteria.

In this research, SysML is applied for Systems Engineering aspects, such as structure definition, stakeholder context, decomposition and requirements representation, UML for software architecture, static behavior and basic dynamic behavior, Petri nets for detailed design and formal verification, and Ontologies for semantics representation. Each language/tool can contribute in modeling some parts of systems, according to its strengths. For instance, SysML is suitable to represent the overall architecture of the system and non-functional properties, such as performance and safety. These properties and terms are specified formally with ontologies; as a matter of fact, domain knowledge is also considered as part of system architecture. Software is modeled with UML, including static behavior and some basic dynamic constraints. Due to the lack of formal definition of UML, for some aspects, such as specific real time requirements, a formal approach is necessary. Complex parts that may need formal verification, such as specific subsystems, UML objects or SysML blocks, are modeled, executed and verified with Petri nets.

The decisions of which tools/languages are applied are given as follows. UML is the *de facto* standard language to model software, being applied for more than 10 years, and during this period, some versions were delivered with important modifications asked by the software industry. In addition, the language is already a compilation of successful methodologies for Object-Oriented Software Engineering that evolves since the 1980's. The newest version, UML 2.1.1, was launched on early 2007. SysML is a language based on UML, sometimes referred as an UML extension for Systems Engineering, or an UML profile. The first version was launched on July, 2006. The use of UML and SysML together seems to be a natural choice, as both languages are sponsored by the OMG group and share common diagrams. UML is a successful standard for modeling software, and SysML was created based on UML but with focus on Systems Engineering.

As SysML and UML are similar, the integration of both system and software models should be made with fewer efforts and collaboration between the various stakeholders will be improved.

Ontologies are widely used in Knowledge and Software Engineering in a variety of applications such as e-commerce and Semantic Web. The benefits of ontologies are that they are well-suited for specification of complex structured domain knowledge, they make explicit the way domain tasks are performed and are independent of the application's implementation language. OWL [27] is one of the main languages to build ontologies. Some other possible applications of ontologies for System and Software Engineering are presented in [26] and [28].

Finally, Petri nets are a formal and graphical tool very suitable to model and analyze several types of systems. There are ways to transform UML dynamic diagrams into Petri nets models, which can be executed. The activity diagram of the new UML version is based on Petri nets, but without a formal basis. Also, it cannot show resource sharing and explicit time constraints. So, activity diagrams can be substituted by Petri nets with advantage. There are a variety of extensions to the initial Petri net model, including the representation of time and data structures.

One of the most important feature of this research is that complexity is treated in several levels. The fact that abstraction, decomposition, and modularization are applied in all levels helps to reduce and manage complexity. In addition, the approach starts at high level and through model transformations achieve more specific models, which are executed (simulated) and proved, in order to guarantee reliability and soundness.

5 First Results and Further Research

The first results of this research were concerned with the application of Petri nets for modeling and analyzing Traffic Signals control and SysML diagrams in the Road Traffic domain.

In [29], it is proposed an approach to model and analyze road intersection traffic signals by means of Petri nets and Linear Logic [30]. A formal proof based on the linear sequent calculus is done in order to analyze the properties of the model for a given scenario. In [31], Petri nets with time associated to transitions were applied in the design of a network of intersections. The approach can prove that unsafe states are not reached, and that desirable states are reached. The results are being extended to be applied to Petri nets with time associated to places for controlling a network of traffic signals. In [32], SysML Requirements diagram are combined with UML Use Cases to improve requirements engineering. System requirements are presented in a graphical and tabular form, and are modeled instead of just written in natural language. Several relationships between requirements, such as decomposition and refinement, are presented with SysML requirements diagram.

The main remaining topics for research are:

- Evaluate the application of System Engineering to Intelligent Transportation Systems.
- Create a process for SysML, and evaluate the advantages and suitability of SysML for complex systems modeling and design.
- Apply SysML and UML to create distributed architectures and models for RTMS.
- Identify and evaluate important properties using Petri nets.
- Create ontologies with OWL for RTMS.
- Create or use UML and SysML profiles for RTMS.
- Evaluate the benefits of Model-Driven Engineering when compared to traditional Requirements-Driven Engineering, in terms of addressing system complexity, reuse and future evolution.

6 Acknowledgement

This work was supported by the Next Generation Infrastructures Foundation and the Research Center Next Generation Infrastructures, both situated in Delft, The Netherlands.

References

1. Rozanski, N., Woods, E.: Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley. (2005)
2. Schmidt, C.D.: Model-driven engineering. *IEEE Computer*. **39** (2006) 41–47.
3. Brown, A.W., Conallen, J., Tropeano, D.: Introduction: Models, modeling, and model-driven architecture (MDA) (Chapter 1). In Beydeda, S., Book, M., Gruhn, V., eds.: *Model-Driven Software Development*. Springer-Verlag., Berlin (2005) 1–16.
4. Magee, C.L., Weck, O.L.: Complex system classification. In Brebbia, C.A., Maksimovic, C., Radojkowic, M., eds.: *Proc. of the Fourteenth Annual International Symposium of the International Council On Systems Engineering (INCOSE)*, Amsterdam (2004)
5. Rouse, W.B.: Engineering complex systems: implications for research in systems engineering. *IEEE Transactions on Systems, Man & Cybernetics - Part C: Applications and Reviews*. **33** (2003) 154–156.
6. Bar-Yam, Y.: When systems engineering fails - toward complex systems engineering. In: *Proc. of the International Conference on Systems, Man & Cybernetics.*, Piscataway, IEEE Press. (2003) 2021–2028.
7. Brooks, F.P.: No silver bullet essence and accidents of software engineering. *Computer*. **20** (1987) 10–19.
8. Jacobson, I., Booch, G., Rumbaugh, J.: *The Unified Software Development Process*. Addison-Wesley. (1999)
9. Balmelli, L., Brown, D., Cantor, M., Mott, M.: Model-driven systems development. *IBM Systems Journal*. **45** (2006) 569–586.
10. Khisty, C., Lall, B.: *Transportation Engineering: An Introduction*. Prentice Hall. (2003)

11. Stough, R.: Intelligent Transport Systems: cases and policies. Cheltenham: Elgar. (2001)
12. Figueiredo, L., Jesus, I., Machado, J., Ferreira, J., Martins de Carvalho, J.: Towards the development of intelligent transportation systems. In: Proc. of the 4th International Conference on Intelligent Transportation Systems., Oakland (2001)
13. National ITS Architecture Team.: Systems Engineering for Intelligent Transportation Systems: An Introduction for Transportation Professionals., Washington, DC. (2007)
14. Kruse, O., Vrancken, J., Soares, M.: Architecture for distributed traffic control: A bottom-up approach for deploying traffic control measures. In: Proc. of the 13th World Congress and Exhibition on Intelligent Transport Systems and Services., London (2006)
15. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture—Practice and Promise. Addison-Wesley Professional. (2003)
16. MOVES: Moves - modeling verification and evolution of software. Technical report, Vrije Universiteit Brussel. (2007)
17. Guelfi, N., Razavi, R., Romanovsky, A., Vandenberghe, S.: DRIP Catalyst: An MDE/MDA method for fault-tolerant distributed software families development. In: Proc. of the OOPSLA & GPCE Workshop on Best Practices for Model Driven Software Development., Vancouver (2004)
18. Coleman, J., Jones, C., Oliver, I., Romanovsky, A., Troubitsyna, E.: RODIN (Rigorous open Development Environment for Complex Systems). In: Proc. of the Fifth European Dependable Computing Conference: EDCC-5 supplementary volume., Budapest (2005) 23–26.
19. Laibinis, L., Troubitsyna, E., Leppnen, S., Lilius, J., Malik, Q.: Formal model-driven development of communicating systems. In Lau, K.K., Banach, R., eds.: Formal Methods and Software Engineering: 7th International Conference on Formal Engineering Methods, ICFEM 2005. Volume 3785 of Lecture Notes in Computer Science., Springer. (2005) 188–203.
20. Kathryn, A.W., Elwin, C.O., Nancy, G.L.: Reusable specification components for model-driven development. In: Proceedings of the International Conference on System Engineering (INCOSE '03). (2003)
21. Lankhorst, M.: Enterprise Architecture at Work: Modeling, Communication, and Analysis. Springer. (2005)
22. Snook, C., Butler, M.: UML-B: Formal modelling and design aided by UML. ACM Transactions on Software Engineering and Methodology. **15** (2006) 92–122.
23. Booch, G., Rumbaugh, J., Jacobson, I.: Unified Modeling Language User Guide, 2nd Edition. Addison-Wesley Professional. (2005)
24. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE. **77** (1989) 541–580.
25. OMG.: SysML final adopted specification. Technical report (2006)
26. Calero, C., Ruiz, F., Piattini, M.: Ontologies for Software Engineering and Software Technology. Springer-Verlag., Berlin, Heidelberg (2006)
27. Lacy, L.W.: OWL: Representing Information Using the Web Ontology Language. Trafford Publishing. (2005)
28. Happel, H.J., Seedorf, S.: Applications of ontologies in software engineering. In: Proc. of the 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE), London (2006)
29. Soares, M., Vrancken, J.: Road Traffic Signals Modeling and Analysis with Petri nets and Linear Logic. In: Proceedings of the 2007 IEEE International Conference on Networking, Sensing and Control (ICNSC 2007), London (2007) 169–174.

30. Girard, J.: Linear logic. *Theoretical Computer Science*. **50** (1987) 1–102.
31. Soares, M., Vrancken, J.: Scenario Analysis of a Network of Traffic Signals Designed with Petri nets. In: *Thirteenth International Conference on Urban Transport and the Environment in the 21st Century*, Coimbra (2007)
32. Soares, M., Vrancken, J.: Requirements Specification and Modeling through SysML. In: *IEEE International Conference on Systems, Man, and Cybernetics (SMC 2007)*, Montreal (2007)

Ordering Functionally Equivalent Software Components

Giovanni Falcone and Colin Atkinson

Chair of Softwareengineering
University of Mannheim, Germany

Abstract. The problem of ordering a set of software components occurs in many application areas. An important case is the problem of ordering the set of components returned in a search, where the results need to be ordered and displayed in a way that is helpful to the requesting user. The dominant ordering criterion is the degree to which components match the functional requirements, but within such a ranking there are often several components which score equally from a purely functional perspective. The challenge addressed in this paper is how to order a set of *functionally equivalent* components based on non-functional criteria. After identifying the different kinds of non-functional characteristics that could be used to distinguish components, and discussing the issues involved in combining them into a single score, we describe the different approaches and technologies that could be used to create a practical implementation of such a ranking technique.

1 Introduction

With the growth of open source code repositories and the recent advent of code search engines to retrieve software from them, the problem of ordering or ranking software component has grown in importance over the last few years. The problem is not limited to ranking search results, but this is by far the most important application. As with all ranking problems, the basic goal is to order entities according to some measure of their “fitness for purpose”. For regular web pages, “fitness of purpose” means the relevance of the content of the web page to the subject the searcher is interested in. As is well known, Google’s pagerank algorithm [BP98] has proven to be one of the most successful measures of this.

In software engineering, the fitness of a software artifact (i.e. a component) for a particular purpose has traditionally involved two factors, the so called *functional requirements* which are concerned with *what* a component does, and *non-functional requirements* which are concerned with *how well* a component does it. Non-functional requirements are therefore often thought of as determining the quality of a component.

Often, the fitness of the functional requirements is given more importance than the fitness of the non-functional requirement, because a component that does the wrong thing is normally of little value even if it does that thing well, whereas a component that does the right thing may still be of some value even

if it does that thing poorly. Sometimes, however, non-functional requirements can be as important as functional requirements in determining the acceptability of a component. For example, if a contractual or legal obligation requires a component to be written in Ada, a component written in another language is not directly “fit for purpose” regardless of how well it meets the functional requirements.

The goal of traditional software engineering methods is, so far as possible, to create components that do exactly the right thing and fulfil all the “must have” requirements. Testing is usually the mechanism used to evaluate the correctness of a component against the functional requirements. If any problems are discovered these are usually removed before the software is shipped.

However, with the advent of software search engines there is now more than one way of obtaining a software component for a particular purpose. As well as developing the component oneself from scratch it has become possible to “harvest” prefabricated components (Components Of The Shelf (COTS)) from specialized software markets or repositories. Most of the specialized search engines available on the web mainly focus on searching for source code like Koders [Kod07], Codase [Inc06], Krugle [Inc07] or GoogleCodeSearch [Lab07]. Currently the only search engines that supports searches for full components based on their interface is Merobase [Mer07].

All of these “engines” for retrieving software components have the same basic problem. When a user presents a query defining the properties he is looking for, the engine needs to find all components that “match” these properties in some way and rank (i.e. order) them according to some objective criteria. In general, there are two fundamental issues involved in ordering a set of software components according to their “fitness for purpose” -

1. ordering them according to their fulfilment of the functional requirements,
2. ordering them according to their quality (i.e. according to non-functional requirements).

In this paper we focus on the latter. We assume that the overall set of search results has been divided into subsets of components, each of which contains components which have the same level of fitness with regard to the functional requirements. The problem that we are addressing in this paper is how to order the members of such a “functionally equivalent” set based on their “quality” (i.e. their fitness for purpose based on non-functional requirements). In other words, we are considering the problem of ordering functionally equivalent components.

The only other ranking approach currently used in a component searching context is the Component Rank approach [Ino03]. This is based on the idea of citation-based ordering as popularized in the PageRank algorithm [BP98] used in an advanced version of Google. The more a component is used by others, the higher the component is ranked. In [IYY⁺05] the authors describe how the ComponentRank approach is used within the Spars-J search environment.

The remainder of this paper is organized as follows. To provide a better understanding of the importance of ordering a set of functionally equivalent software

components, we start with a general description of the problem area in the context of a large-scale software component search engine. In the subsequent section we analyze different property sets related to software components and explain in general how these can be used for ordering components. We also discuss different possible ways of calculating a single ranking value in order to simplify the ordering of the entries in a result set.

2 Functional Equivalent Sets

In the introduction we already mentioned the two fundamental issues involved in ranking the software components discovered by a search engine. To provide a better understanding of the problem area, in this section we explain how functionally equivalent sets of software components can be obtained from a component search engine. We use merobase.com for the example, since it provides the best support for component-oriented searches, but the discussion is applicable to other code search engines as well. We close this section by defining the term *functionally equivalent* more precisely.

2.1 Creating Functional Equivalent Sets

The easiest way of defining a search query is by identifying the topic of interest through simple keywords, as is well known from text-based search engines like Google. Generally speaking, the better the description of the topic, the smaller the number of documents likely to be added to the result set and the greater the likelihood that they will fit the user's purpose.

In contrast to textual documents, software documents possess a defined structure based on the grammar of the used programming language. For software components, it is therefore possible to use programming abstractions such as classes and methods etc., to support interface-oriented searches [Mer07] as well as simple text-based searches. These essentially represent two extremes in the range of search queries that can be used to drive component searches.

Text-based queries usually contain a small number of topic-describing keywords which can easily be refined if the entities in the result set do not correspond to the users needs or the size of the result set is too large. Interface-oriented queries, on the other hand, define the specific properties the user is looking for and thus usually yields small results sets since only components that exactly fulfil the given requirements are returned. This kind of query is aimed at supporting reuse within mainstream software development projects as envisaged, for example, by Sommerville [Som04]. The software developer defines the overall software architecture and the stubs of some of the application parts. These stubs can then be directly used as the basis for interface-oriented search requests, and if needed the user can redefine the application stubs or use a less precise request to retrieve a larger number of software components.

2.2 Definitions on Functional Equivalent Sets

Depending on the precision of a search request, the number of entries in a result set can vary from a small number to a very large number. Regardless of how the query is defined, however, the entities in the final result set should conform to the given request to some degree. For interface-based searches, the entities should closely match the criteria given by the query. However, the more precise the description of the interface that the user is searching for, the lower the number of matching components usually found. In the extreme case, the result set might be empty even when there are components that fulfil the user's general requirements to a lesser degree of conformance. Therefore, merobase implements a so called multi-phase searching algorithm where entities with less than 100% conformance are added to the result set until a sufficient number of candidates have been found. Each phase of this multi-phase search process produces a set of results which build a subset of the final result set. Since each successive subset has a lower level of conformance to the query than its predecessor, there is a natural ordering of these subsets within the final set according to their level of conformance.

The elements within a given subset are effectively functionally equivalent (i.e. have the same level of functional conformance) so additional non-functional information needs to be taken into account to order them internally. By ordering the subsets based on their level of conformance to the query defined by the user, and order the components within the subsets based on non-functional properties, an overall ordering of the result set is achieved.

3 Using Non-functional Requirements for Ordering

In the previous section we described how a software search engine can be used to obtain a set of reusable software components. An initial ordering is achieved by grouping the overall set of available components into subsets, each with the same degree of functional conformance to the query. However, since these subsets can themselves contain several components, an internal ordering is needed within each subset. Since this cannot use functional information, this has to be based on non-functional properties of the components. Before we elaborate on the details of how this is done, in the following section we analyze the different classes of information that could be used for this purpose.

3.1 Orderable versus Non-Orderable Properties

Some non-functional properties of software components are inherently discrete and non-orderable. A good example is the language that a component is written in. There is no universal ordering between languages (e.g. Ada is better than C is better than Java etc.). This is usually regarded as a black and white (i.e. binary) property. Either a component has the property or it does not. Of course, a particular user may well define his/her own hierarchy. Other properties are

inherently orderable, such as the delivered bandwidth etc. It is also important to note that binary acceptance of a component can also be thought of as an ordering problem (i.e. those that are accepted are ranked before those that are not).

3.2 Static versus Dynamic

Considering both source code components and binary components, possible information used for ordering can be retrieved either from a static analysis of the component, and in the case of binary components, also by executing them with predefined test cases. This is the idea that underpins the Extreme Harvesting approach [HA04]. There are various kinds of information that can be gathered from static analysis of software components but they generally fall into three subcategories: *code related information*, *certification information* and *metadata* (i.e. author, company etc.).

Code Related Informations By *code related information* we mean metrics that can be obtained by a static analysis of the software. In the literature, there is a huge set of different software metrics. However, there are a few well known groups of metric suites which are of particular importance like the Halstead metrics [Hal77], the McCabe metrics [McC76] or the so called OOMetrics [CK94] for object oriented software. Other metrics are also known and widely used. The huge set of available metrics is not of immediate value in helping distinguish components in a result set, as these often describe different facets of the software. This means that even if we choose a small subset of code related information, we need to combine the values to a single one in order to achieve a final ordering. In the following section we describe possible ways of combining the different kinds of data derivable from components, not just metrics in the classic sense.

First we need to analyze which kinds of metrics are suitable for characterizing a component. In general, a metric is useful if it has a large range of numeric values. Each metric describes a certain facet of a component, even if some of them often describe the same facet in a slightly different way. Therefore, one of the main challenges is choosing the best set of metrics to describe components from different point of views.

Static metrics can be complemented by information gained when the component is executed. This is possible if a component is directly available in a binary form or can be transformed to a binary form by compilation. This is not always straightforward, since a component can often depend on other components which themselves depend on other components etc. leading to quite complex dependency structures. However, the problems involved in executing components in the general case are not the focus of this work and are not considered further.

The information that can be gained from the execution of a component, such as the quality of service, may describe additional useful aspects of the component. Usually, however, they provide more information about the same underlying characteristics as the static metrics but in a more accurate form.

Certification In addition to the code related information some components are certified by third parties and therefore we would expect these to be of higher quality, even if an analysis of the code related information would lead to them being regarded as being equivalent. Using the whole web as a possible component repository, only a small subset of components might be certified. At present, therefore, certification is only usable as auxiliary information.

Meta Data Beside the code related data, describing the *quality* (static analysis) or the *quality of service* (dynamic analysis) of a component, other information not directly related to the code may be derived. We group this kind of information under the term *metadata*. Examples are the components origin, author, license or the position of keywords within the content etc.

The components origin and the position of keywords within the content are both used in text-based document searches. In the latter case, the PageRank algorithm uses this kind of information directly by giving text a higher relevance if it is found at a particular position like a header. The component's origin is used indirectly in the form of the URL from which the document can be obtained. Both kinds of information might also be used in the ordering of software components. In the case of keywords, names are of higher importance if they play particular roles like being a class-name or a method-name, or if they are found in a particular place e.g. in a method body. In contrast to text-based searching, where the URL of a document is regarded as an important factor in their "fitness of purpose" relative to the given keywords, this is of minor importance for software components. However, the origin of a component might play an important role if the organization which developed it has a good reputation. For example, components provided as commercial products by big players in the software market might be regarded as being of higher relevance than those from smaller, lesser known organizations. However, this is highly subjective and is related to the problem of having non orderable properties.

In addition to its origin, the type of license associated with a component is often of interest to users. Since the number of different license types is not particularly large, one might expect that an absolute ordering of the license types might be the easiest way of using this information in the ranking process. However, from our experience the relevance of license type for ordering the result of a component search is highly dependant on the kind of software under development. For example, a developer of commercial code might only accept a license type like the *GNU Lesser General Public License (LGPL)* or the *Apache License* and others would be of minor value. In contrast, a developer of open source code might accept also other licensing types.

In general, the choice of information to use as the basis for an ordering plays a significant role in the quality of the ranking algorithm for software components used in large scale component search engines. Once the choice has been made, the next challenge is to develop an optimal algorithm for combining the information into a single score that can be used as the basis for the overall ordering of components. This is discussed in the following section. Since such an individual

score can satisfy different needs, we first give a short overview of the different ways a single score can be perceived (i.e. can be used as a customized measure of the user's needs)

4 Representative Scores

Analyzing the different ways of combining the raw information discussed above into a single score for ordering software components we observe two extremes. At one extreme, each individual user might have his own understanding of the properties that should be taken into account to order the components and what their relative importance should be. At the other extreme, we could define one universal ordering which can be used as the default when the user has no wish for one customized to his own needs. The best choice might lie somewhere in between these extremes, since we think that the most important influence on the algorithm used to calculate scores is the application domain, rather than the wishes of individual users. In other words, we believe that users within a given domain will have very similar perceptions of ranking requirements, while users from different domains will differ more widely in their view (see Figure 1).

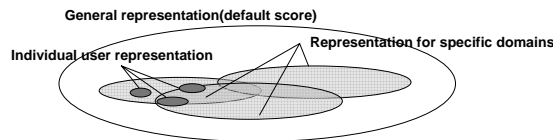


Fig. 1. The extremes of representative scores

5 Calculating a single Score

In the previous sections we described what kind of information related to a component can be used as the basis for an ordering. However, in order to compare two or more components on the basis of their relevance to a given request, a single score needs to characterize a component's overall *quality*. In the following subsection we present an effective way of calculating a single universal score, neglecting for the moment the problem of defining individual-customized scores or domain specific scores.

5.1 Weighted sub-scores

A simple but effective way of calculating a single value from a set of characteristic values is to assign each value a weighting according to its importance in the overall quality measure. However, this way of calculating a single score has some

major drawbacks, because the individual weights need to be known in advance of the calculation. Since the weights can only be reasonably determined from knowledge of the significance of the characteristics used to calculate a single score, adapting them to the needs of a specific user or domain is only possible with major effort. Another drawback arises if the quality of a single characteristic does not vary in a linear way (is not a linearly increasing or decreasing quality) but instead is characterized by a compound function which can only be determined with major effort.

A simple example is given by the *lines of code* (*LOC*) metric which is widely used in software engineering. In the case of a programming language like Java we find a correlation between the metric and the type of the component. For example, interfaces by nature have a smaller number of lines of code than regular class files. If this metric is used as the basis for differentiation and we search for software components rather than interfaces, we observe that starting at 0 *lines of code* the general relevance increases. Once a certain threshold is achieved we observe, in turn, that the quality decreases again. Figure 2 illustrates this observation schematically. In the case of such non-linear behaviour it is no longer possible to use simple weighting factors for each characteristic value. It is still possible to use functions as weighting parameters, but these might only be calculatable with major effort and possibly only by specialists.

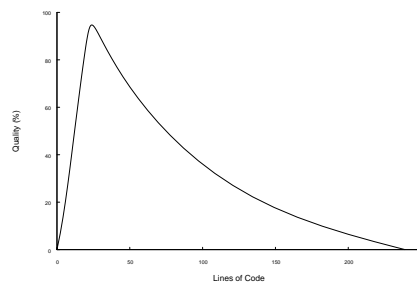


Fig. 2. Non-Linear relation between Quality and LOC

A simple weighting based approach does not therefore seem to be a good basis for calculating a single universal score from the raw characteristics of a software component. Therefore, to provide a general and user friendly way of calculating such scores we need to consider other approaches.

5.2 Neural Networks

The usage of neural networks (i.e. a rule based calculation of a universal score) allows a significant simplification over the previous described approach. The importance of each characteristic in the final score is calculated in a simple rule based way. Additionally, neural networks can be trained with training values in

which the rules, input and outputs sets are automatically modified and adapted to the training values. This allows even complicated functions that may arise in the consideration of characteristic input values to be represented easily without further knowledge.

User feedback on orderings provides enough training data to improve the network if needed. As we have seen in the previous section, the domain plays a significant role for an ordering. Therefore a set of rule bases can be provided for the different domains.

However, the use of neural networks still has some drawbacks. The membership to a given set can only be true or false, which is problematic if we use numeric intervals as the basis for the subsets, where the values at the boundaries of the intervals are treated equally to the values inside and are mapped to the same value in the result set. This would wipe out relative quality scores of components. A solution to this problem is to use a range of values to measure a component's quality score and to characterize its overall quality using fuzzy sets[Zad65].

5.3 Neuro Fuzzy Systems

Neuro Fuzzy systems are an enhancement of standard neural networks with the notion of fuzzy sets, as these provide the properties given in the previous sections [Lip06]. The use of a neuro fuzzy system to calculate a quality score on the basis of the characteristic values would consist of at least three levels: an input level, a rule level in the middle and an output level.

As the basis for the input level the characteristic values of a software component are used. These are mapped to the output set according to their membership in the input fuzzy sets and by using one or more rules from the rule level. If a Mamdani [MA99] controller is used, the output value is still fuzzified. The final score can therefore easily be obtained by defuzzification of the output level (see figure 3).

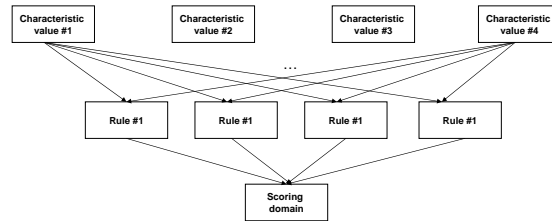


Fig. 3. Neural net representing the neuron levels

The major drawback with this method of calculating a quality-based score for software components is the computational performance. However, the advantages of this method in terms of the possibility of online adaptation to a user's needs

easily outweigh this disadvantage. The performance problems are reducible if a default score is used which can be calculated before a search query is processed.

6 Conclusions and Future Work

Starting with a set of functional equivalent software components we discussed the issues involved in establishing an ordering based on non-functional properties. To this end we discussed which values could usefully characterize a component and how one could derive a single value out of these under different circumstances. As we explained, when a simple weighting based approach is not suitable, there is a need to allow users to adapt the scoring technique on-the-fly without any detailed knowledge about the ordering mechanism itself. We discussed the advantages of using neural networks and in particular neuro fuzzy systems which allow a fuzzy treatment of the characteristic values. In order to realize these mechanisms we are currently analyzing the relative importance of the discussed characteristic values with a view to identifying the optimum weighting factors.

References

- [BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.
- [CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
- [HA04] Oliver Hummel and Colin Atkinson. Extreme harvesting: Test driven discovery and reuse of software components. In *IRI*, pages 66–72, 2004.
- [Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.
- [Inc06] Codase Inc. Codase - Source Code Search Engine. <http://www.codase.com>, 2006.
- [Inc07] Krugle Inc. Krugle- Code Search for Developers. <http://www.krugle.com>, 2006-2007.
- [IYY⁺05] Katsuro Inoue, Reishi Yokomori, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto. Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering*, 31(3):213–225, 2005.
- [Kod07] Koders. Koders - Source Code Search Engine. <http://www.koders.com>, 2007.
- [Lab07] Google Labs. Google Code Search. <http://www.google.com/codesearch>, 2007.
- [Lip06] Wolfram-Manfred Lippe. *Soft-Computing*. Springer, Berlin [u.a.], 2006.
- [MA99] E. H. Mamdani and S. Assilian. An experiment in linguistic synthesis with a fuzzy logic controller. *Int. J. Hum.-Comput. Stud.*, 51(2):135–147, 1999.
- [McC76] McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.
- [Mer07] Merotronics. Merobase - the Component finder. <http://www.merobase.com>, 2006-2007.
- [Som04] Ian Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.
- [Zad65] Lotfi A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965.